Integrated Mixed-Signal Solutions

Customizing Flash Players Using SigmaTel SDK 2.4xx

Version 1.2

September 26, 2003

COMPANY CONFIDENTIAL 9/26/03

5-3xxx-HG13-1.0-091203

Copyright © 2003 SigmaTel, Inc. All rights reserved.

All contents of this document are protected by copyright law and may not be reproduced without the express written consent of SigmaTel, Inc.

SigmaTel, the SigmaTel logo, and combinations thereof are registered trademarks of SigmaTel, Inc. Other product names used in this publication are for identification purposes only and may be trademarks or registered trademarks of their respective companies. The contents of this document are provided in connection with SigmaTel, Inc. products. SigmaTel, Inc. has made best efforts to ensure that the information contained herein is accurate and reliable. However, SigmaTel, Inc. makes no warranties, express or implied, as to the accuracy or completeness of the contents of this publication and is providing this publication "AS IS". SigmaTel, Inc. reserves the right to make changes to specifications and product descriptions at any time without notice, and to discontinue or make changes to its products at any time without notice. SigmaTel, Inc. does not assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation special, consequential, or incidental damages.

1. TABLE OF CONTENTS

1. TABLE OF CONTENTS 2
1.1. List of Figures 7
1.2. List of Tables 7
2. REVISION HISTORY 9
3. SCOPE 10
3.1. Using This Document 10
4. SYSTEM OVERVIEW 11
4.1. ROM Bootloader 11
4.1.1. Example Boot Sequence 11
4.2. Firmware Files 12
4.2.1. bootmanager.sb 12
4.2.2. usbmsc.sb 13
4.2.3. stmpsys.sb 13
4.2.4. resource.bin 13
4.3. Recovery Mode 13
4.3.1. USB Boot Mode Class 13
4.3.2. Example Recovery Mode Boot Sequence 14
4.4. Host Device Drivers 14
4.4.1. Custom Host Device Drivers 14
4.4.2. Minimum Host Requirements: 14
5. FEATURE OVERVIEW 15
5.1. SDK2.400 15
5.2. SDK2.410 -- Current Release 15
5.3. SDK2.42x -- Future Release 15
6. MEMORY MANAGEMENT 16
6.1. Resources 16
6.1.1. Module Overlays 16
6.1.2. Menu Codebanks 17
6.1.3. Funclets 17
6.1.4. Application Overlays 17
6.1.5. Media Device Drivers 17
6.1.6. Bitmaps 17
6.1.7. Fonts 18
6.2. Memory Map 18
6.2.1. ROM Memory 18
6.2.2. Bootloader Memory 19
6.2.3. DCC Memory 19
6.2.4. Player Memory 20
6.3. Resource Loading 20
6.3.1. SysLoadResource 20
6.3.2. SysCallFunclet 20
6.3.3. SysCallFunction 20
7. PLAYER START UP 21
7.1. Hardware Init 21
7.1.1. USBMSC Hardware Init 21
7.2. Headphones 22
7.3. Restore Saved Settings 22
7.4. Module Init 22
7.5. Media Error Checking------FSInit() 22
7.5.1. External Media Considerations 23
8. REAL TIME OPERATING SYSTEM 24
8.1. Module Processing 24
8.2. Executive Loop 25
8.2.1. Execute Modules 26
8.3. Menu Manager 29
8.3.1. SysCallFunction 29

COMPANY CONFIDENTIAL 9/26/03

8.3.2. SysWaitOnEvent 29
8.4. ISRS 30
8.5. Brownout Strategy 30
8.5.1. False Brownout (IMPORTANT!) 30
8.5.2. Core Voltage Brownout 31
8.5.3. Battery Voltage Brownout 31
8.5.4. Vddio Voltage Brownout 32
8.5.5. Headphone Short Strategy 32
8.6. USB Brownout Implementation (USBMSC) 32
9. MODULE MESSAGING COMPONENTS 33
9.1. Module Message Definition (File: msgequ.inc) 33
9.2. SysPostMessage 33
9.2.1. SysPostMessage Assembly Example Usage 34
9.2.2. FSysPostMessage C Example Usage 34
9.3. Module Message Queue Descriptor 34
9.3.1. Example Message Queue Descriptor 35
9.4. Router Table 35
9.5. Module Table 35
9.5.1. Example Module Table 37
10. USER INTERFACE 38
10.1. Button Basics 38
10.1.1. Button Event Descriptions 38
10.1.2. Button Event Detection 38
10.1.3. Button Event Processing 40
10.1.4. Scan Matrix Button Definitions 40
10.1.4.1. Scan Matrix GPIO Connections 40
10.1.4.2. Scan Matrix Button Definition 40
10.1.5. Non-Scan Button Definitions 41
10.1.5.1. Non-scan Button Connections 41
10.1.5.2. Non-scan Button Events 41
10.1.6. Hold Button Assignment 41
10.1.7. Button Event Assignments 41
10.1.7.1. Single Button Event Mapping 42
10.1.7.2. Double Button Event Mapping 42
10.2. Remote Control/Single Wire Button Example 42
10.3. Jog Dial Example 44
10.3.1. Jog Dial Buttons Events 44
10.3.2. Jog Dial Code Modifications 44
10.4. Rotary encoder 45
10.4.1. Device driver code 45
10.4.2. Encoder absolute position 45
10.4.3. Encoder Initialization 45
10.4.3.1. GPIO pins selection 45
10.4.3.2. Define the number of phases per detents 45
10.4.4. Encoder Application Interface 45
10.4.4.1. UI Callback Mechanism 46
10.4.4.2. Encoder Position Change Notification 46
10.4.5. Test Examples 46
10.4.6. Application Example 46
10.5. Graphical LCD 46
10.5.1. LCD Initialization 46
10.5.2. Display Resources 47
10.5.3. Bitmaps 47
10.5.4. Strings 48
10.5.5. Fonts 48
10.6. Displaying Strings 48
10.7. Contrast 48
10.8. Backlight 48

10.9. Graphical LCD Messaging Module API 49
10.9.1. Framebuffer and LCD 52
10.9.1.1. Module 52
10.9.1.2. Software Abstraction Layer (SAL) 52
10.9.1.3. Hardware Abstraction Layer (HAL) 53
10.9.1.4. Messages 53
10.9.1.5. Framebuffer Details 53
10.9.2. Resize the LCD Display 53
10.9.2.1. Changes for USB LCD interface 54
10.9.2.2. Changes for player LCD interface 54
10.9.2.3. The speed of display on LCD 54
10.9.3. LED GPIO assignment 55
10.9.4. LedTable entry 55
10.9.5. LED Module 55
10.9.6. LED initialization 56
10.9.7. LED Module Messaging API 56
10.9.8. Example LED Messages 56
10.9.9. Example LED flashing Messages 57
10.9.9.1. Right to left flashing example 57
10.9.9.2. Left to right flashing example 57
10.9.9.3. Moving out flashing example 57
10.9.9.4. Moving in flashing example 57
11. FEATURE SET 58
11.1. Track Playback Operation 58
11.1.1. Play/Pause 58
11.1.2. Stop 58
11.1.3. FF 58
11.1.4. RW 59
11.1.5. Variable FFWD & RWND Scan Speed 59
11.1.6. A/B Mode 60
11.1.6.1. AB Decoder Messages 60
11.1.6.2. AB Status Flags 61
11.1.7. Volume Control 61
11.1.7.1. SDK2.1xx Multi-Stage Volume Control 61
11.1.7.2. SDK 2.0xx Single Stage Mixer Volume Control 61
11.1.8. PlayerLib Extracted Track Information 62
11.1.8.1. Track Name 62
11.1.8.2. Track Metadata Information 62
11.1.8.3. Track Number 62
11.1.9. Decoder Extracted Track Information 63
11.1.9.1. Encoder Track Time 63
11.1.9.2. Decoder Track Time 63
11.1.9.3. Track Encoding Properties 63
11.2. Playset 63
11.3. Play Mode 64
11.4. Voice, FM, and Line-In Record 64
11.4.1. Record Settings 64
11.4.2. Stmp Hardware Setup 65
11.4.3. Encoder Module Setup 66
11.5. GEQ 66
11.5.1. GEQ Message Definitions 68
11.5.2. GEQ Initialization 68
11.5.3. SDK GEQ Example Usage 68
11.6. Delete Files 69
11.7. FM Tuner 69
11.7.1. FM Tuner Module Messages 70
11.7.2. FMTuner Global variables: 71
11.7.3. Example FM Tuner UI 71
11.7.4. Customizing the FM Tuner 71
11.7.5. Improving FM Tuner Performance 72

12. PLAYLIST API 73
12.1. PlayList Data Structure 73
12.2. Required PlayList Library Entry Points 73
12.2.1. Playlist_GetCurrentSongFile() 73
12.2.2. Playlist_GetNextSongFile() 73
12.2.3. Playlist_GetPreviousSongFile() 73
12.3. Optional PlayList Library Entry Points 74
12.3.1. Playlist_Init() 74
12.3.2. Playlist_GetSongFileByNumber() 74
12.3.3. Playlist_SetSongByNumber() 74
12.3.4. Playlist_SetMode() 74
12.4. Player Library 74
12.4.1. Goto Track 74
12.4.2. PlayerLib_SetState() 75
12.4.3. PlayerLib_FastForward() 76
12.4.4. PlayerLib_SetSongName() 76
12.4.5. PlayerLib_SetSongName() 76
12.4.6. PlayerLib_SkipToPreviousSong() 76
12.4.7. PlayerLib_EnablePlaylist() 76
12.5. PlayList Implementation 76
12.6. Playlist Customization 82
13. METADATA AND STRING HANDLING 86
13.1. MetaData Support 86
13.1.1. MP3 ID3V2 86
13.1.2. MP3 ID3V1 86
13.1.3. WMA Metadata 86
13.1.4. WAV Metadata 86
13.2. Metadata overview 86
13.2.1. MP3 Metadata 86
13.2.2. Get Current Song Type 87
13.2.3. Customize MP3 Metadata Display 88
13.2.4. WMA MetaData 89
13.2.5. WAV Metadata 89
13.3. String handling and Localization Overview 89
13.4. Long File Names 90
13.4.1. Playlist_LFNGetFileName 90
13.4.2. ChangeDirtoFileEntryDir 91
13.4.3. FSLFNGetFileName 91
14. IO LIBRARY 92
14.1. _find functions 92
14.1.1. _findfirst() 92
14.1.2. _findnext() 92
14.1.3. _findclose() 92
14.2. Size Functions 92
14.2.1. GetDiskFreeSpace() 92
15. CONTROLLING THE HARDWARE 93
15.1. Multi-stage Volume Control 93
15.1.1. Multi-Stage Volume Control Theory of Operation 93
15.1.2. Multi-Stage Volume Control API 93
15.1.3. Reading the current volume level 95
15.2. PLL/ Voltage 95
15.2.1. Basic Algorithm For Adjusting PLL 95
15.2.2. User Level Requirements For PLL Control 95
15.3. Battery Monitoring 96
15.3.1. Rechargeable LION 96
15.3.2. LRADC Sampling API 96
15.3.3. Battery Voltage Reporting API 96
15.3.4. SDK Battery Monitoring Example Implementation 96
15.3.5. USB Battery Monitoring (USBMSC) 97

15.4. ADC Output 97
15.5. DAC Output 97
16. PLAYER SHUTDOWN 98
16.1. Shutdown Events 98
16.1.1. Timed Auto Shutdown 98
16.1.2. User Shutdown 98
16.1.3. Exception due to Brownout or Headphone Short 98
16.2. Save Settings 98
16.2.1. Battery Level Dependency 99
16.2.2. Saving Variables 99
16.2.3. Settings.dat format 99
16.3. Headphone Shutdown 100
16.4. RTC Shutdown 100
16.5. Power Off 100
17. SYSTEM MESSAGING MODULES 101
17.1. Soft Timers (stmp3xxx_sdk\system\msgmodules\software\softtimer\softtimers.c) 101
17.1.1. Softtimer Message Definitions 101
17.1.2. Example Softtimer Messages 101
17.2. Parsers 102
17.3. Decoders 103
17.3.1. Decoder Message Definitions 103
17.3.2. Decoder Functions Specific To Each Module 106
17.3.3. Common Decoder Functions Used By Decoder Modules 107
17.3.4. DecoderSR / DecoderCSR Flag Definition 107
17.4. Encoders 108
17.4.1. Encoder Message Definitions. 108
17.4.2. EncoderSR / EncoderCSR Bits 108
17.5. Mixer 108
17.5.1. Mixer Message Definitions 108
18. FOREGROUND PROCESSING CHAIN 111
19. FILE SYSTEM / MEDIA 112
19.1. Multi-Nand 112
19.1.1. Hardware requirements: 112
19.1.2. Build option 112
19.1.3. Initial Configuration and Setup 112
19.2. File System 113
19.2.1. FileSystem API 113
19.2.2. FAT 113
20. USBMSC 114
20.1. Display customization 114
20.1.1. USBLCDDisplayInit 115
20.1.2. USBLCDIdle 115
20.1.3. USBLCDReading 115
20.1.4. USBLCDWriting 115
20.1.5. USBLCDLowBattery 115
20.1.6. USB Throughput 115
20.1.7. USB Battery Display 116
20.1.8. Changing Battery Resources 116
20.2. GROUP6 VENDOR SPECIFIC SCSI COMMAND SUPPORT 116
20.2.1. Data Structures 117
20.2.1.1. SCSI Command Data Structure 117
20.2.2. Adding SCSI Commands 117
20.2.3. Adding OpCodes 118
20.2.4. Adding Updater Commands 118
20.2.5. Adding Custom Bitfields to Command Structure 119
20.2.6. Command Handling with GetCustomerExtentionSCSIHandler() 119
20.2.6.1. Method 1 - Return UpdaterFunction via SCSI Handler 119
20.2.6.2. ....Method 2 - SCSI Handler as Updater Function 120
20.2.7. Adding Updater Functions 121

COMPANY CONFIDENTIAL 9/26/03

APPENDIX A. BUILDING BLOCKS 122
A-1. Build Execution 122
A-2. Player Output Files 122
A-3. Creating Resources 122
A-4. Mixed Assembly / C 122
A-5. Locator Labels : F_lc_ 122
APPENDIX B. ADDING ANOTHER MENU MODULE 123
APPENDIX C. ADDING A NEW MENU CODEBANK 126
APPENDIX D. ADDING A NEW MODULE 128
D-1. Module Table (modulemsg.asm): 128
D-2. Add to the MakeFile (player.mak): 128
D-3. Add to the descriptor file (stmp3400.dsc): 128
APPENDIX E. BUILDING CUSTOM FONT RESOURCES 129

1.1. List of Figures

Figure 1. Code Overlays 16
Figure 2. ROM Memory 18
Figure 3. Bootloader Memory 19
Figure 4. DCC Memory 19
Figure 5. Player Memory 20
Figure 6. Player Init 21
Figure 7. System Architecture Block Diagram 24
Figure 8. Messaging Modules 25
Figure 9. Main Executive Loop 26
Figure 10. Execute Modules Flowchart 28
Figure 11. Example Button Event Timing Diagram 38
Figure 12. Button Processing Flowchart 39
Figure 13. 43
Figure 14. 43
Figure 15. 48
Figure 16. 59
Figure 17. FFWD/RWND Event Timing diagram 60
Figure 18. AB State Machine 61
Figure 19. GEQ Control 67
Figure 20. Example GEQ filters 67
Figure 21. 78
Figure 22. 79
Figure 23. 80
Figure 24. 81
Figure 25. 81
Figure 26. 85
Figure 27. Foreground Processing Chain 111
Figure 28. Flow Chart of Updater SCSI Command 118

1.2. List of Tables

Table 1. Revision History 9
Table 2. Module Event Signal/Wait Control Bits 27
Table 3. Message Definition 33
Table 4. Message Data Structure 33
Table 5. Module Message Queue Descriptor 34
Table 6. Module Table Entry Structure 36
Table 7. SYSTEM_LCD_INIT_SEQ RESOURCE 47
Table 8. Graphical LCD Messaging Module API 49
Table 9. LED Messages 56
Table 10. 60
Table 11. MUSIC PLAYMODES 64
Table 12. GEQ Messages 68
Table 13. FM Tuner Module Messages 70
Table 14. LcdExample FMtuner User Interface 71

COMPANY CONFIDENTIAL 9/26/03

Table 15. No tag selected (default). 88
Table 16. ID3V1 tag selected. 88
Table 17. ID3V2 tag selected. 88
Table 18. Unpacked String Example 90
Table 19. DBCS/Packed String Example 90
Table 20. Compiler Packed String Example 90
Table 21. Multi-Stage Volume Project Specific Constants 94
Table 22. Multi-Stage Volume Definitions 94
Table 23. Volume Stages 94
Table 24. Softtimer Messages 101
Table 25. Parser messages initiated by the decoder 102
Table 26. Parser messages called from menu level 102
Table 27. Decoder Messages 103
Table 28. DecoderSR/DecoderCSR Bits Parser 107
Table 29. Encoder Messages 108
Table 30. EncoderSR/EncoderCSR Bits Parser 108
Table 31. Mixer Messages 108

2. REVISION HISTORY

Revision No.	Date	Change Description
1.2	9/26/2003	Appendix E -- Using custom fonts require changes in SDK2.4xx
1.1	9/11/2003	Various updates and corrections in the following sections: nands supported, lcd, VBR decoding, Appendix A & E
1.0	8/4/2003	Initial Release of SDK2.400

Table 1. Revision History

3. SCOPE

This document is intended to be used by firmware developers using the STMP34xxx family of devices in conjunction with the SigmaTel SDK.

This document contains high level overview of various customizable features. Please refer to the extranet for Application Notes that provide more details on certain subjects.

SDK Versions relevant to this doc:

- STMP3410/1342 SDK version 2.4xx

3.1. Using This Document

The SigmaTel Extranet is referenced throughout the document. Hyperlinks are used in the document to link to the referenced document on page. The links will only work correctly if the Extranet login information is saved as a cookie otherwise the login page will be accessed.

Click here to access the reference document on the SigmaTel Extranet.

Code snippets in the document use the following paragraph/font style:

```
// this is a C comment
int i=0;
```

Variable names are reference by the symbol name. If accessible in ‘C’ will be called FVar otherwise is Var.

Function names are denoted by Foo() to distinguish between variables. The name reference is not intended to match the actual prototype.

4. SYSTEM OVERVIEW

Flash players using the SigmaTel solution require at least 1 NAND flash to store firmware on the device (refer to Section 4.2. Firmware Files for a list of files stored on the flash). The size of the firmware that resides on flash is dependent upon the project. SDK lcdexample requires ~1Mbyte of the player flash to store firmware and resources.

SigmaTel firmware relies heavily on resources that are stored on flash and loaded into RAM at runtime. Referred to as *overlays*.

Several NANDs can be “combined” to appear as a single device to the end user -- referred to as *MNAND (Multi-NAND)*.

In addition, several removable flash device types are supported and all SDK examples can be built for either NAND only or MMC using the command line build option.

Note: Refer to Section 5. Feature Overview for the SDK version that will support each feature.

- MMC

  MMC device drivers use the SPI port as the physical transport layer and support protocol layers defined by: MultiMediaCard System Specification Versions 1.4, 2.11, and 3.1

- SD - MMC compatible mode only (no encryption)

  SD device drivers only support MMC compatibility through the SPI physical transport layer according to the SD Memory Card Specification Part 1 Physical Layer Specification Version 1.0 and the protocol layer as defined by SD Memory Card Specification Part 2 File System Specification Version 1.0.

4.1. ROM Bootloader

ROM will always execute upon a device reset and will enable the boot device specified by the bootmode hardware configuration. Supported boot devices are I2C, SPI, NAND, and USB. If an error occurs during the ROM boot, USB boot is attempted. USB boot mode will time-out after about 5 minutes. USB boot will try to enumerate a driver using SigmaTel VID and PID embedded in the ROM, thus the recovery mode driver or equivalent is required to be installed in order to actually boot over USB.

The ROM forces a USB boot when the boot device does not contain valid firmware. There is also a hardware/software combination called *Recovery Mode* that allows users to invoke the USB bootmode overriding the hardware configuration. NAND bootmode is the standard boot method supported by the SDKs. The code in bootmanager.sb supports the boot process from the NAND flash.

Regardless of the boot mode, the code loaded by ROM is encrypted by the updater. If another bootmode is used (i.e. I2C EEPROM) the bootloader file must be encrypted before programming the EEPROM. Contact SigmaTel to obtain a utility to encrypt the boot load file.

4.1.1. Example Boot Sequence

Assumes hardware is configured for nand bootmode.

- Power-on -- pswitch active level achieved
- ROM loads code (bootmanager.sb) stored in reserved system blocks from NAND per bootmode configuration.

- bootmanager samples USB connection:
  - if USB CONNECTED: loads usbmsc.sb from reserved NAND blocks
  - if NO USB connection: loads stmpsys.sb reserved NAND blocks
- Reboot if USB disconnected or connected

4.2. Firmware Files

The firmware files are written to hidden blocks marked with a special code in the NAND redundant area during a firmware update.

The host updater will copy the following files to flash when a STMP3410/1342 device is detected during update.

- bootmanager.sb
- usbmsc.sb
- stmpsys.sb
- resource.bin

Note: All 4 files are required in order to properly install the host device drivers.

4.2.1. bootmanager.sb

Applicable for NAND boot modes only. Loaded by ROM during a successful nand boot, this code will determine if USB is connected and load the appropriate firmware into RAM. It is not loaded during recovery mode or other non-nand bootmodes.

If another boot device is required (I2C or SPI) code must be written to support a similar operation using the alternate boot device.

After bootmanager loads the firmware into RAM it jumps to the reset vector. All code that is loaded by bootmanager, must have a valid reset vector in order to startup.

4.2.2. usbmsc.sb

Loaded by bootmanager when USB is not connected at startup. This file contains the software that interfaces to the native USB Mass Storage drivers on the OS.

The USBMSC firmware uses SCSI commands. It operates at the device sector level and does not contain a filesystem or have any knowledge about files being transferred.

4.2.3. stmpsys.sb

Loaded by bootmanager when USB is not connected at startup. This contains all system code used by the player. System code is considered to be all functions resident in static memory (not overlays).

When USB is connected during player operation, the device is reset and bootmanager loads USBMSC from flash if USB is still connected when bootmanager executes.

4.2.4. resource.bin

Resources loaded into RAM as needed during player operation by SysLoadResource (system code contained in stmpsys.sb). This file contains code/data overlays, fonts, and bitmaps.

4.3. Recovery Mode

Recovery mode is detected by ROM and either PSWITCH or PLAY inputs can trigger recovery mode. The input trigger method is defined by the hardware bootmode configuration.

Refer to D-Major Audio Decoder datasheet on the SigmaTel Extranet for details on configuring the bootmode pins for the recovery mode trigger.

During ROM boot, recovery mode is checked by sampling the selected input. If the input is active for >5 seconds the ROM forces USB Boot Mode Class.

This boot method is required to recover from a flash corruption. USB boot mode class will be automatically entered if no valid firmware is found on the boot device. For initial flashing of NANDs (i.e. manufacturing lines) USB Boot Mode is entered upon power-up if USB is connected.

4.3.1. USB Boot Mode Class

When USB Boot Mode is selected, the Boot ROM code sends the generic Product ID over USB to the host and the host then loads the Player Recovery Class Driver, which is able to communicate with the Boot ROM code. When the updater is run, the player recovery device driver transfers the usbmsc.sb file, contained in the host installation directory over USB handing it over to the Boot ROM code, which loads the usbmsc.sb into the device RAM and resets the Program Counter (PC) to the beginning of the usbmsc code. The drives are then accessible and an updater appears, which allows the user to burn the player binaries (stmpsys.sb, usbmsc.sb, resource.bin, bootmanager.sb) into flash.

When USB boot mode is selected through the Boot Mode Switches (M1-M6), the same behavior as recovery mode is performed without the hardware input. .

4.3.2. Example Recovery Mode Boot Sequence

Refer to the Development Platform User Manual on the SigmaTel Extranet for more detailed instructions on using Recovery Mode with the NEW UniqueID method.

Assumes configured for nand bootmode.

- Power-on -- pswitch active level achieved
- ROM detects recovery mode (input active >5sec) or USB Boot Class is initiated
- ROM enumerates Player Recovery Device Class device driver using a fixed PID
- When executed, the firmware updater utility loads the usbmsc.sb onto the player
- Once usbmsc.sb is transferred over USB, the player will re-enumerate in USB Mass Storage mode.
- The removeable drive(s) will appear in my computer at this time.
- The Updater GUI application will appear at this time.
- Clicking the Start button within the updater GUI will begin the firmware update.
- If flash was successfully updated with working firmware, the player will boot using the method described above (ROM->booty->stmpsys)

4.4. Host Device Drivers

The host drivers are different for the external media types. Nand only hosts must use firmware files built for NAND only players and MMC hosts must contain firmware files built for MMC players.

4.4.1. Custom Host Device Drivers

SigmaTel will customize the device drivers including strings and bitmaps for end customer's products. The customer is responsible for updating the usbmsc to match the USB VID/PIDs used in the customized host.

Use the Submission Form for Customized USBMSC Host Driver on the SigmaTel Extranet to request a customized host package for your company.

The latest SigmaTel versions of the host installation disks with working SDK lcdexample binaries can be obtained from the extranet. These hosts will work with the USB IDs and strings provided in SDK examples.

The host installation drivers must contain all 4 binaries in order to install without errors.

4.4.2. Minimum Host Requirements:

OSes supported: Win98SE (USBMSC driver installation required), WinMe, Win2K, WinXP

5. FEATURE OVERVIEW

5.1. SDK2.400

STMP3410 lcdexample & ledexample

STMP1342 ledplayer support only

New features include:

- Type 2 NAND only
- Win98 USBMSC driver -- eliminates need for DCC
- New Updater
- Support for Windows formatter (non-SSFDC formats)

5.2. SDK2.410 -- Current Release

STMP3410 lcdexample & ledexample

STMP1342 ledplayer support only

New features include:

- Multi-NAND -- Type2 only
- MMC

5.3. SDK2.42x -- Future Release

STMP3410 lcdexample & ledexample

STMP1342 ledplayer support only

- Full support of both Type1 & Type2 NANDs in single of multiple configurations
- DRM support
- Speed optimizations
- FAT32 support
- Mac OSX support with Type2 NANDs **could be released independently if no firmware changes.

COMPANY CONFIDENTIAL 9/26/03

6. MEMORY MANAGEMENT

6.1. Resources

All resources are stored in resource.bin on NAND flash in SigmaTel SDK examples. The creation of the resource file occurs at build-time and is controlled by resource.inc.

Executive functions only load resources in P memory space. If resources are required in X or Y memory space the module in P is responsible for loading the other resources (the decoder/encoder module is an example of this and loads P, X & Y in application space).

No resource information is stored back to flash before reloading another resource, so static memory requirements must not be a part of the module resource but be located in static/system memory.

6.1.1. Module Overlays

Modules are functional blocks of code that execute in round-robin fashion during player operation. The executive controls loading each module as determined by the real-time operating system. The modules are controlled by inter-module communication and operating system controls.

Message queues for each module are part of the operating system and are resident in static memory.

During customizations, additional messaging modules can be added as system resources allow.

The organization in memory of the modules is important. The first word of the module overlay must be the resource number. This location is used by the executive to determine which module is loaded in memory. It is also used in the debugger to match the correct symbols with the loaded code. The organization of the menus is different than the other modules. The beginning and end of module space is labeled using Tasking locator labels so the resource can be loaded into memory.

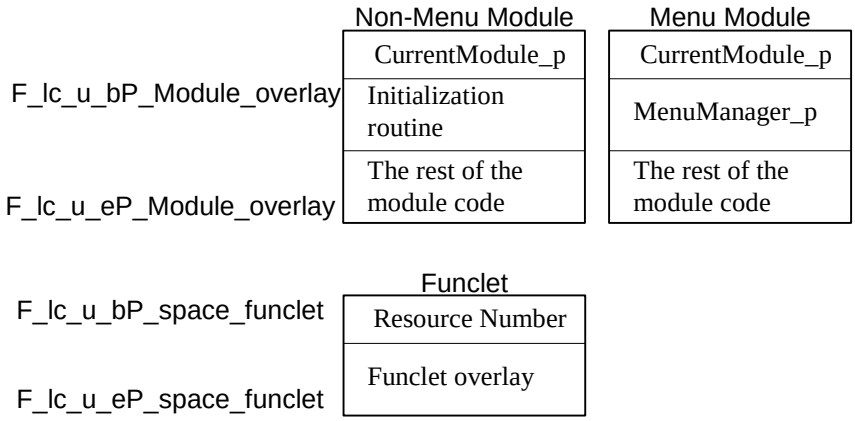


Figure 1. Code Overlays

COMPANY CONFIDENTIAL 9/26/03

6.1.2. Menu Codebanks

A menu codebank is actually a module that is loaded by the executive. A menu module passes control to another menu module via the menu manager. The menu manager code exists in each menu codebank resource.

This is the primary region available for customizing user interface operation and the player statemachine(s).

6.1.3. Funclets

Funclets are individual functions that are overlaid in the same memory region. A portion of the function remains in static memory and calls SysLoadFunclet() to load the appropriate resource.

Funclet memory space is a way of expanding system memory by allowing functions to overlay a portion of static memory space. Funclet calls can only be made from the user level and typically are seldom used functions that do not have critical timing considerations.

Funclets can be nested X levels. Where X is a configurable constant defined in project.inc. Two (2) words of Y memory are used for each nested funclet.

```
; maximum number of nested funclets
MAX_NESTED_FUNCLET   equ     6
```

Funclets must also follow a particular memory organization similar to modules. The resource number must be the first word in memory. Similar to modules, the resource number is used by SysCallFunclet and also allows symbolic debugging of overlaid code.

6.1.4. Application Overlays

Application overlays are additional memory used by modules. This memory space should not be considered for use during customizations. Loading the application space is controlled by modules NOT the executive and can be easily corrupted if custom modules do not properly force reloading the application space.

Application space is used by decoder and encoder modules as well as various filesystem routines.

6.1.5. Media Device Drivers

Different device drivers are used for different media accesses. The overlaid device drivers make use of a “media mapper” to load the appropriate media driver overlay into memory.

The Media Map Table defines which media device driver to load for each device number in the system. Current implementation does not allow both device drivers to be used at the same time. Refer to Section 19.3. Media Mapper for more details.

6.1.6. Bitmaps

Bitmaps are converted to resources using a buildtime utility. Bitmaps are stored vertically in the resources. Only black and white bitmaps, 1 bit per pixel are supported.

If the SigmaTel LCD module is used during the customization, the programmer simply adds the bitmaps to the build and uses the resource number in lcd messages to display the bitmap.

6.1.7. Fonts

Detailed instructions for creating custom font resources are included in Appendix APPENDIX E. Building Custom Font Resources. SigmaTel has various scripts that can be used to convert bitmaps of each character into the resource files.

The font bitmaps must conform to certain guidelines to be used in the SigmaTel localized fonts solution.

- A DBCS codepage translating DBCS to Unicode must be referenced.
- Each character must be contained in a individual bitmap.
- Each character must be named **fontprefix_xxxx.bmp**, where **xxxx** is the unicode value of the character in hex and **fontprefix** typically corresponds to the codepage name.

6.2. Memory Map

Because of the ROM functionality the lowest addressable P RAM should not be used in .s-records loaded by the bootloader. The first 512 words of X RAM should also not be loaded by the bootloader.

The initial portion of the memory map is used to load code and cannot be overwritten until the bootloader sequence is complete. The SDKs reserve the portion of memory used by ROM and booty for overlaid code which is only present in resource.bin and not the s-records.

6.2.1. ROM Memory

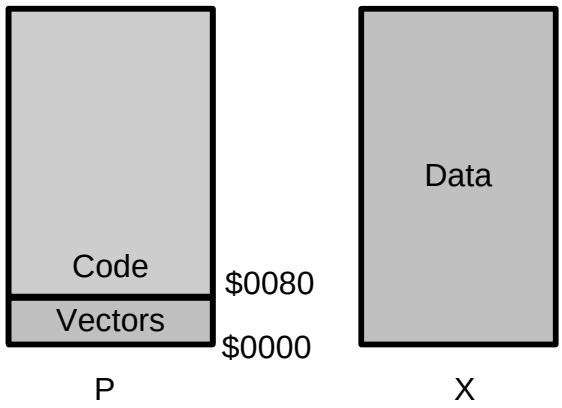


Figure 2. ROM Memory

6.2.2. Bootloader Memory

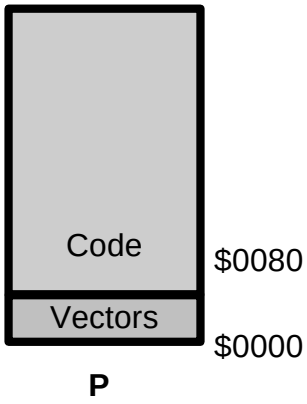


Figure 3. Bootloader Memory

6.2.3. DCC Memory

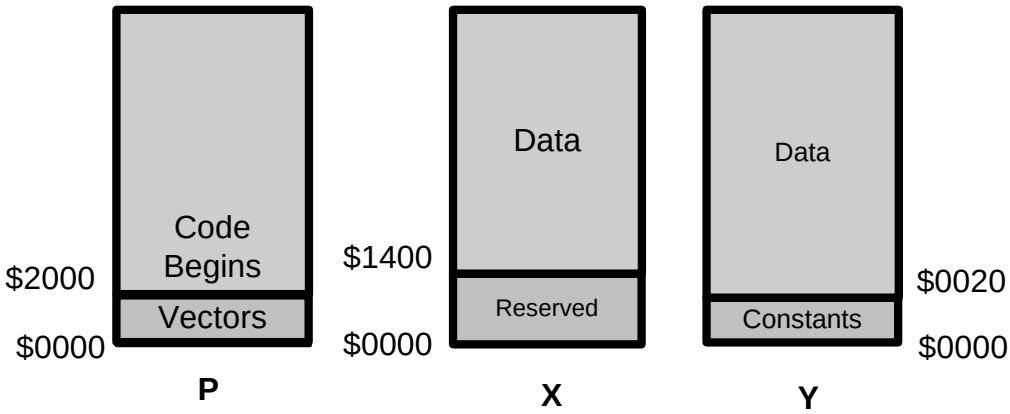


Figure 4. DCC Memory

6.2.4. Player Memory

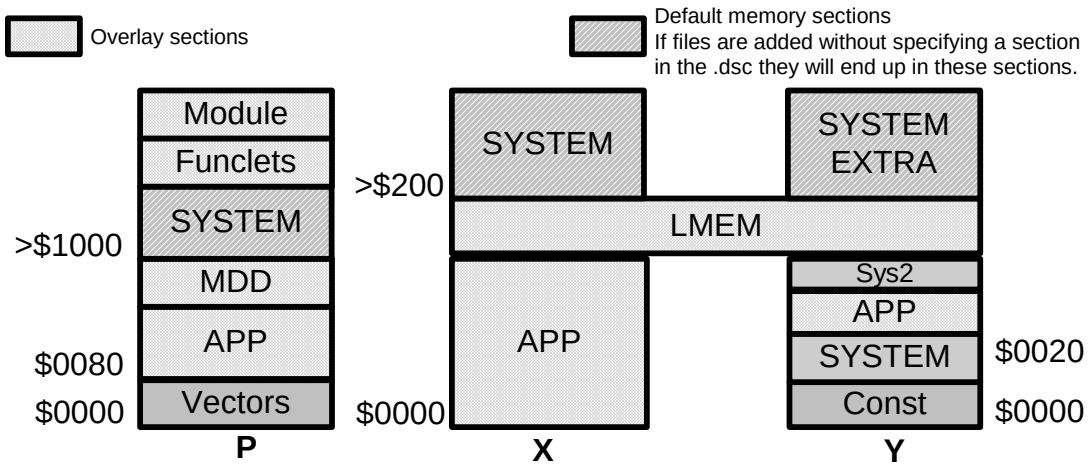


Figure 5. Player Memory

6.3. Resource Loading

6.3.1. SysLoadResource

SysLoadResource() is called to load any resources from resource.bin to RAM. It uses a "light" version of the filesystem in order to speed up accesses.

This function is called from various places in the system and loads code, data, bitmaps into RAM.

6.3.2. SysCallFunclet

Each funclet consists of a few words in system space that call SysCallFunclet() to load the function. SysCallFunclet() in turn calls SysLoadResource() to load the funclet resource into RAM.

6.3.3. SysCallFunction

SysCallFunction() is part of the Menu Manager API. It allows menu operations to span different codebanks. Using this function allows executing a function from another menu codebank then returning back to the calling codebank.

In the SDK 2.2xx implementation SysCallFunction() calls BLOCK other modules from running each time there is a menu codebank switch.

7. PLAYER START UP

main.asm and mainmenu.c provide player initialization in the SDK examples. Some initialization operation is required to execute in a specific order and should not be changed.

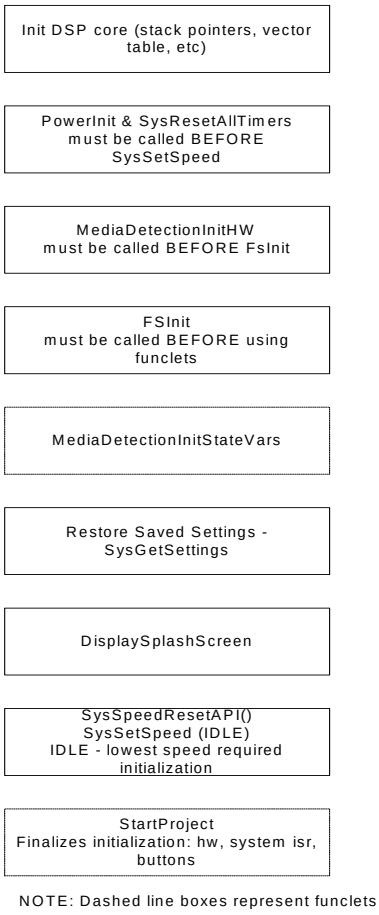

Init DSP core (stack pointers, vector table, etc)
PowerInit & SysResetAllTimers must be called BEFORE SysSetSpeed
MediaDetectionInitHW must be called BEFORE FsInit
FSInit must be called BEFORE using funclets
MediaDetectionInitStateVars
Restore Saved Settings - SysGetSettings
DisplaySplashScreen
SysSpeedResetAPI() SysSetSpeed (IDLE) IDLE - lowest speed required initialization
StartProject Finalizes initialization: hw, system isr, buttons
NOTE: Dashed line boxes represent funclets

Figure 6. Player Init

7.1. Hardware Init

SigmaTel provides a recommended chip initialization in PowerInit and AnalogInit. The DC/DC converter is initialized and various other chip and peripheral setup is performed in StartProject function in main.asm.

7.1.1. USBMSC Hardware Init

The PowerInit and AnalogInit is also used for the USBMSC to reduce power consumption. They are called during start-up. There are some differences from the player, which can be identified with the ‘PLAYER’ build option.

7.2. Headphones

In order to reduce headphone pop during power-on in a timely manner, SigmaTel has developed a system level algorithm. SigmaTel strongly recommends that this algorithm be used during initialization.

The headphone amp is powered up in ClassA mode and the DAC is used to ramp a signal from ground to Vag. PowerUpHeadPhones is called from StartProject in main.asm (system\common\sysheadphones.c).

The initial prefill value should put the DAC output at ground. The temporary DAC isr in that file refills the DAC buffer with values that cause the output to ramp to the mid point (Vag).

This can be monitored on a scope by looking at the LOL/LOR output pins. Pay close attention to the beginning and end of the ramp to make sure everything is smooth.

If the power rails don't completely discharge (takes several seconds on engineering board) before a power cycle, there will be a pop. The headphone amp is ramped to ground before power down in the SDK to prevent a quick power cycle causing a power on pop. For shutdown, refer to Section 16.3. Headphone Shutdown.

7.3. Restore Saved Settings

SysGetSettings() is called during main.asm to restore the system variables stored in settings.dat. SysGetSettings() sets up the memory overlays required to execute SysLoadSettings().

All saved variables should be initialized with default values for cases when settings.dat is removed.

Any module messages that are required to initialize to the system to the default settings are sent during initialization in mainmenu.c. For example, EQ and PlayMode saved settings must be initialized by sending a module message.

7.4. Module Init

The executive calls an initialization routine for each module prior to normal module processing if the initialization signal bit (EVENT_INIT) is set in the EventSignal field of the module table.

Some modules may also require project specific input via a message which should be sent during initialization (typically in mainmenu.c).

For example, the GEQ module requires an initialization message which configures the filter Q and center frequency of the 5 bandpass filters.

7.5. Media Error Checking------FSInit()

Fsinit() returns carry clear if there is no error or set if there is error found. The accumulator A1 has error bits set. The global variable g_FSInitErrorCode also has the same error bits as A1. NAND is device 0 (even if MNAND), and external media (SD/MMC) is device 1. For the external device, no error code is returned and the carry bit is clear if the media is not inserted.

The error bits for internal device only, no external device players:

- Bit 0 - set if device 0 is not installed or invalid.
- Bit 1 - set if there is error in the bootsector of device 0
- Bit 2 - set if there is error in the FAT table of device 0
- Bit 3 - set if there is error in the directory of device 0

- Bit 4 - cleared if device 0 has supported formatted media

For two device players (NAND and SD/MMC):

- Bit 0 - set if device 1 is not installed or invalid.
- Bit 1 - set if there is error in the bootsector of device 1
- Bit 2 - set if there is error in the FAT table of device 1
- Bit 3 - set if there is error in the directory of device 1
- Bit 4 - cleared if device 1 has supported formatted media
- Bit 5 - set if device 0 is not installed or invalid.
- Bit 6 - set if there is error in the bootsector of device 0
- Bit 7 - set if there is error in the FAT table of device 0
- Bit 8 - set if there is error in the directory of device 0
- Bit 9 - cleared if device 0 has supported formatted media

7.5.1. External Media Considerations

If external media is not present it looks the same as an invalid media. If player has external media, you should use NUM_REMOVABLE_MEDIA (in project.inc) and bit 0 to decide the external media state. If external media is inserted and bit0 is set, display the media error.

The SDK implements 3 different error messages for players with external media to differentiate the possible scenarios on the LCDexample:

- FSInit returns bit1, 2, or 3 set then "External media error! Remove and reinsert, or reformat the media if the problem is persistent."
- FSInit returns bit 4 set then "Error! Reformat external media with Sigmatel formatter."
- FSInit returns bit 5, 6, 7, 8, or 9 set, "Error! Reformat internal media with Sigmatel formatter."

Only a single error message will displayed if the build only supports one device:

- FSInit returns bit 1, 2, 3, 4, or 5 set, "Error! Reformat internal media with Sigmatel formatter."

8. REAL TIME OPERATING SYSTEM

The player operating system consists of a main executive loop, functional modules with intermodule communication via messaging, and interrupt processing.

Memory management considerations are an integral part of the operating system architecture. The module layer is completely overlaid, such that only a single module is resident in memory at a given time. All the other blocks in Figure 7 System Architecture Block Diagram can be called anytime (funclets are technically overlaid, but are exceptions in this sense).

The menu modules provide project specific user interfaces and control functional operation of the player. The menu module is the only module that is signaled when button inputs are received.

The system modules provide a messaging interface with the menus providing system feedback to the user level. In addition there is some amount of messaging between system modules.

The rest of the system functions are available in static memory and can be used by any module.

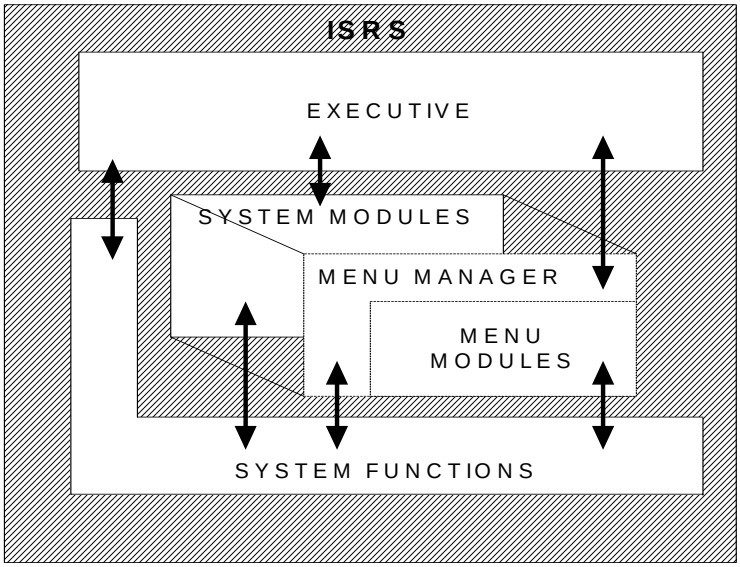


Figure 7. System Architecture Block Diagram

8.1. Module Processing

ExecuteModules() will process each module in order of the module table as determined by the Event Wait and Event Signal control flags. As each module is processed it may post messages to other modules. If the new messages are posted to a module that has already been run, the new messages will not be processed until the rest of the executive runs and the module is called the next time through the loop.

The module controls it's own callback by returning the Event Wait control word to the executive upon module completion. All modules can be signalled by either a message or timer. In addition, the menu module will be called back when a button event is received.

Each module code and data memory is overlaid on each other. In addition, there may be multiple types of each module, for example menus, decoders, etc. Switching between dif-

COMPANY CONFIDENTIAL 9/26/03

ferent module types (i.e. decoders) occurs by overwriting the resource number in the module table with the resource number of new decoder, etc. The executive will load the new module next time through the module table.

The module table in the SDK examples consists of the following modules in this order:

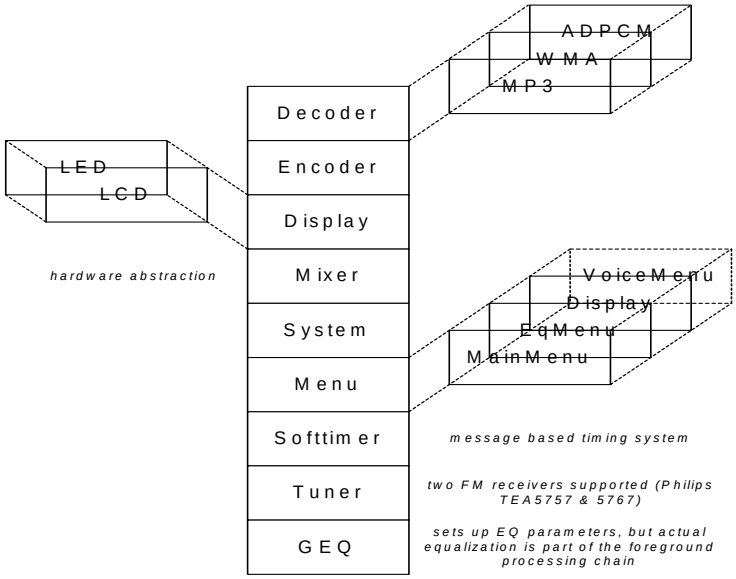


Figure 8. Messaging Modules

8.2. Executive Loop

The executive loop (main.asm) provides background processing control of the system. The following functions are performed by the main executive loop:

- Headphone short processing -- qualification & possible shutdown only. Detection is provide via interrupts.
- USB detection, a system reboot is initiated so the host connectivity software can be loaded. USB detection is a polled function.
- External media state change detection -- polled function. Determine if the state of the media has changed since the last time through. Menus will process.
- Module processing initiated
- Check for buttons and boost system speed if buttons pending or reduce speed if truly idle -- SetExecState and ExecSpeedAndBtnCheck

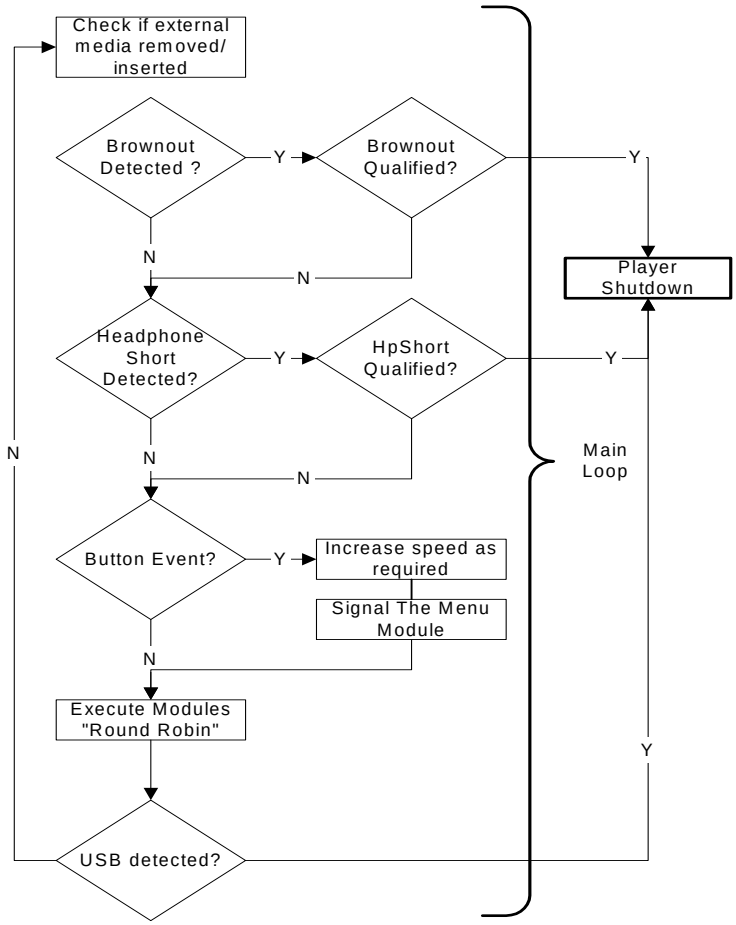


Figure 9. Main Executive Loop

8.2.1. Execute Modules

ExecuteModules() starts at the top of the module table and determines in order which modules to execute by using the EVENT SIGNAL and EVENT WAIT fields in the module table. Typically the module is executed when a condition exists in EVENT SIGNAL that is allowed by EVENT WAIT.

EVENT WAIT is a control word for the callback request and is defined each time a module exits and typically masks the EVENT SIGNAL word to provide blocking. For example, just because a message is in the queue, the module can refuse to be called back until a timer expires.

The system assembly routine SignalEvent() can be used to set the EVENT SIGNAL for any module from any module. DecoderForceInit() is an example of a routine that uses SignalEvent() to reinitialize the decoder module.

Bit	Module Signals	Event Signal	Event Wait
0	EVENT_NONE	0: Do not force a call to this module. 1: Force a call to this module.	0: Force call masked. 1: Force call unmasked.
1	EVENT_MESSAGE	0: No message has been sent to this module. 1: A message has been sent to this module.	0: Do not call if messages are in queue. 1: Call if messages are in queue.
2	EVENT_TIMER	Not used.	0: Module does not have an active timer. 1: Module has an active timer.
3	EVENT_BUTTON	0: No Button Event has occurred. 1: Button Event has occurred.	0: Do not respond to Button Events. 1: Respond to Button Events.
4	EVENT_ BACKGROUND	Not Used	0: Do not process background routine 1: Process background routine
5	EVENT_REPEAT	0: Do not repeat this module 1: immediately repeat this module	Not Used
7	EVENT_INIT	0: Do not run the init routine for this module. 1: Run the init routine for this module, and also forces the module to run immediately after the init.	Not used.

Table 2. Module Event Signal/Wait Control Bits

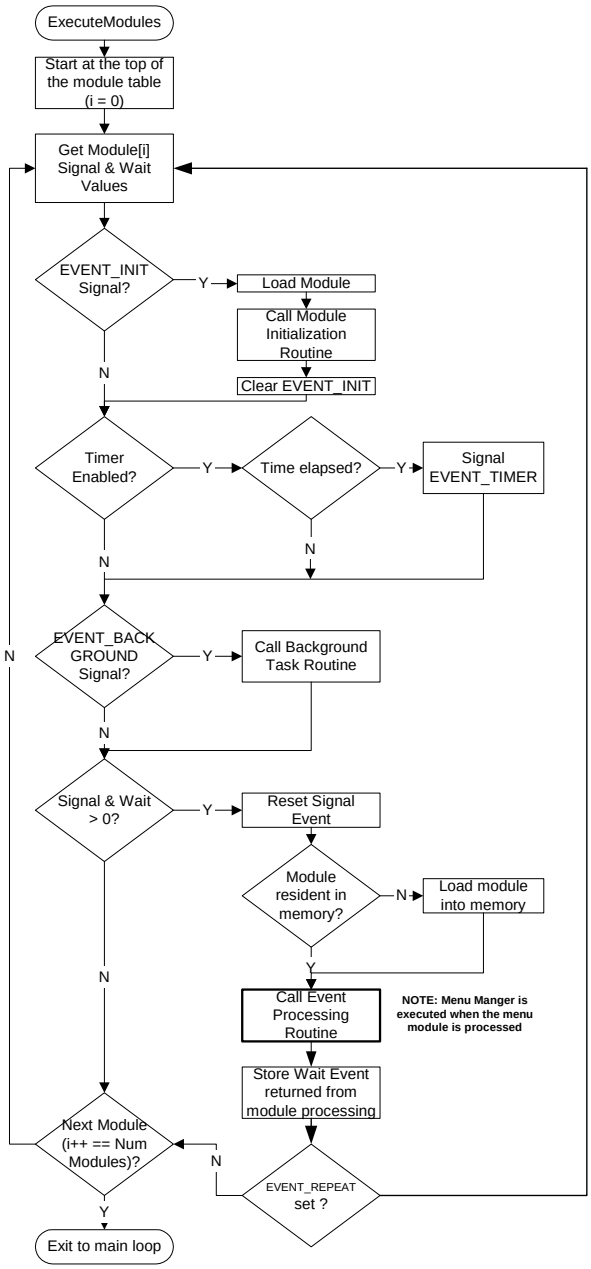


Figure 10. Execute Modules Flowchart

8.3. Menu Manager

The menu manager provides the interface between the executive and the menu codebanks. The menu manager is actually resident in each menu codebank. It handles swapping stacks and context switches between codebanks as well as the module table manipulation allowing the executive to load the proper codebank the next time through ExecuteModules(). The codebanks are actually nested with the UserTask function being the first codebank level.

The menu manager also provides two functions to the menu level that allow the menu to switch codebanks and return to the executive.

8.3.1. SysCallFunction

This function allows one menu codebank to execute a function in another menu codebank and return to the next line of code in the first menu just as any ordinary function call.

The normal executive is used with the EVENT_REPEAT flag to load and execute the new codebank.

```
int _asmfunc SysCallFunction(unsigned int RESOURCE,int
_reentrant (int,int,int*),  int, int, int *);
```

The following prototype is required for all menu functions that need to be called from another codebank.

```
int d _reentrant Foo(int a, int b, int c);
```

To be called as follows:

```
d = SysCallFunction(RSRC_FOO_MENU_CODE_BANK,Foo,a,b,c);
// next line of code
```

The menu codebank containing Foo is loaded, Foo(a,b,c) is executed and program execution continues on next line when Foo returns.

8.3.2. SysWaitOnEvent

This function allows a return to the executive and sets the menu callback. The menu manager will handle setting up the interface to the executive to enable the proper SIGNAL WAIT control flags and timer setup. Either a button event, message, or expired timer can be the callback. If a button or message callback is requested, the EVENT_REPEAT flag will be used to immediately callback the menu if buttons or messages are pending; otherwise the remaining modules will process.

The function prototype:

```
int return _asmfunc SysWaitOnEvent(unsigned int
uEvent,struct CMessage *,int uLength);
```

return = callback event type

uEvent = desired callback event (EVENT_MESSAGE or EVENT_BUTTON or either event EVENT_MESSAGE|EVENT_BUTTON)

CMessage* EventInfo = typically referenced by the union EventTypes (gEventInfo) and contains either the entire message or button event number. The menu manager determines which message is next in the queue and loads it into this structure and advances the head pointer.

uLength = timer in mS. If -1 no timer callback is activated. The menu manager sets up timer values in the module table and the EVENT WAIT flag (EVENT_TIMER) as required by this field.

In this code snippet, no callback events are registered, no pointer is required since no events will be returned, 0mS timer is started. This effectively allows the executive to continue until the next menu module opportunity.

```
SysWaitOnEvent(0,0,0);
```

This call is the typical entry/exit point to the executive. When a message or button is received the menu will begin execution after this call. The menu will be repeatedly called back when menu messages or button are queued.

```
iEvent = SysWaitOnEvent(EVENT_MESSAGE|EVENT_BUTTON,&gEv-
entInfo.msg,-1);
```

8.4. ISRS

The following interrupts are enabled in the SDK examples:

- Brownout (IRQB)
- HeadphoneShort (IRQA)
- DAC/ADC (level 3) : moves data from DMA buffer to en/decoder source buffer and handles overflows.
- Timers :
  - Decoder/Encoder algorithmic processing moves data from the en/decoder source buffer to the sink buffer.
  - button timer : polls the buttons and lradc (remote buttons and battery)
  - mS timer : mS tick

8.5. Brownout Strategy

“Brownout” refers to the condition of a system voltage level dropping below a threshold that could lead to an uncontrolled reset or shutdown. STMP devices have internal detection methods that will cause a brownout indication via IRQB.

STMP3410/1342 supports multiple voltage source brownout detection circuits as follows. In addition each one is handled differently in software. Upon an IRQB assertion, the various brownout conditions are tested to determine the functionality to execute to maximize battery life.

Some brownout routines shut down very quickly (such as the battery fallout case) and can not protect against flash write failures. These flash write failures can lead to corrupted media, which requires a format to operate properly.

8.5.1. False Brownout (IMPORTANT!)

False brownout has been a common problem with players that have a larger than expected ripple on the power supply lines. The current Vddd and Vddio brownout settings are approximately 100mV below the actual voltage level. If the ripple is large enough, a false brownout could occur leading to unwanted player shutdowns and even flash corruption if the event occurred when writing to the flash (such as voice recording and writing settings.dat).

Note: It is very important for developers to ensure that the brownout settings are well above any typical noise that might be seen on Vddd and Vddio. Poor layouts will have noise on the power supplies and brownout levels will have to be increased thus reducing battery life. It is strongly suggested that layouts be submitted to mp3quality@sigmatel.com for review prior to production to maximize battery life.

8.5.2. Core Voltage Brownout

The core voltage (Vddd) is controlled through the DC/DC Converter. Droops in the Vddd are typically affected by a headphone short or a low battery condition.

Vddd brownout voltage threshold level is controlled through the HW_DCDC_VDDD register in the STMP device and is set in *sysspeed.inc* to correspond to approximately 100mV lower than the core voltage level set for each speed index.

Vddd brownout is enabled in PowerInit() during StartProject() initialization and changed each time the voltage is changed during *sysspeed()* execution.

Note: See Section 8.5.5. Headphone Short Strategy for more information.

During a headphone short the headphone amp will draw enough current to cause a droop in the core voltage while in Class AB mode.

- Upon a Vddd brownout condition, put the headphone amp into class A mode because it will draw negligible current without startup issues associated with a complete shutdown. If the headphone short is causing the core voltage droop, it will return to normal voltage levels when the headphone amp is in Class A mode.
- Compare the long term battery voltage average against BATT_BO_CNTS as defined in project.inc to determine if the battery level has returned to normal indicating a headphone short caused the brownout. Otherwise if the battery level is still low, the battery life is most likely ending so a controlled shutdown will occur including SysRTCShutdown().

Note: BATT_BO_CNTS is a project-specific variable that can be adjusted per project. SDK examples use the equivalent of 850mV as the threshold because the LRADC values are typically low due to inaccuracies and noise in the STMP device.

- The headphone is returned back to Class AB mode, if the headphone short is the cause of the voltage droop. The ISR will trigger again immediately if the short is still present, thus no audio will be heard during a headphone short but the player will not shutdown. *** CHANGE -- ClassAB is returned too fast in this version ***

8.5.3. Battery Voltage Brownout

The battery voltage (Vbat) brownout detection is monitored through the Low Resolution Analog to Digital Converter (LRADC). This brownout is intended to handle the “battery fall out” condition where the battery is physically disconnected.

The brownout threshold is setup during player initialization by the following call, where `BATT_BROWNOUT` is defined in *project.inc* using constants define in *regs3410.h*. SDK examples use 830mV as the comparator threshold and BATT_BO_CNTS of 850mV (again due to LRADC inaccuracies and noise in the STMP device).

```
SysLRADCBrownoutInit((WORD)(BATT_BROWNOUT));
```

The battery fallout case is qualified from a quick disconnect & reconnection by quickly sampling and averaging the battery voltage using the raw LRADC samples instead of the button isr sampling. LRADC update frequency is considered in the averaging loop.

Assuming a 68uF cap on battery line (dt/dv = 13.6uS/0.1V) as defined in the modular reference schematics, the voltage will be held up for ~13.6uS by the cap.

In the battery drop out case, the crystal can be affected by battery droops before the Vddd brownout will trip so the 13.6uS shutdown is met in the SDK examples assuming worst case speed index of IDLE (10MHz).

The RTCShutdown is bypassed in the battery brownout case because it typically takes 500uS to execute. The battery is either slowly dying or has fallen out, so the RTC will be corrupted anyway and it is more important to stop execution before the crystal loses power rather than try to save RTC.

Refer to comments in the code for more information.

CAUTION: Hard real-time requirements of cycle counting are implemented in this routine making it very sensitive to change in the lowest speed index and initially executed isr code.

8.5.4. Vddio Voltage Brownout

The I/O voltage (Vddio) brownout detection is controlled through the DC/DC and monitors the I/O voltage level.

The Vddio brownout voltage threshold level is controlled through the HW_DCDC_VDDIO register in the STMP device. The core brownout level is enabled in PowerInit() during StartProject() initialization.

If a VDDIO brownout is detected, the player shutdown without qualification calling Sys-RTCShutdown().

8.5.5. Headphone Short Strategy

The headphone short detection circuitry does not work correctly in the STMP3400. The core brownout detection handles shutting down the player in this condition.

STMP3410/1342 do have a headphone short detection circuitry that is enabled. However there is a problem with the implementation that is related to the hardware design. When a song with a strong bass line is played at full volume into 16-Ohm headphones (strongly clipped output), the headphones will periodically shut down due to a falsely detected short circuit.

In SDK2.320, the headphone short detection circuitry is only used to qualify the core brownout condition to correct the false detection.

8.6. USB Brownout Implementation (USBMSC)

USB brownout protection has been implemented for the USBMSC in SDK2.400. The USBMSC will call PowerInit() during start-up to set the voltage and brown out levels for Vddd and Vddio. Also, the SysLRADCBrownoutInit() will be called to initialize battery brownout.

When an IRQB occurs, the UsbSysBrownOutIsr() routine will process the interrupt. This routine is similar to the player, but was customized for USB operations. For example, the headphone short strategy for core voltage brownout has been removed since it is not needed when the USBMSC is running. The number of battery samples was also modified to match the 48-MHz clock for USB connectivit.

COMPANY CONFIDENTIAL 9/26/03

9. MODULE MESSAGING COMPONENTS

All module components are considered project-specific and are defined in the file *modulemsg.asm*.

The message definitions are in *msgequ.inc*.

9.1. Module Message Definition (File: *msgequ.inc*)

Message Ids - each module has a unique message id.

Message Definition - the messages for each module have a unique value consisting of the associated module ID and a sequential message number. The sequence of messages must match the order of the jump table in the module ProcessQueue function. The first and last message must also be defined for error processing.

The Parser message type is now routed to the menus.

MSGID	Bits 23-16	Bits 15-0
00	MSG_TYPE_DECODER	Decoder message number
01	MSG_TYPE_ENCODER	Encoder message number
02	MSG_TYPE_PARSER	Parser message number
03	MSG_TYPE_LCD	LCD message number
04	MSG_TYPE_MIXER	Mixer message number
05	MSG_TYPE_SYSTEM	System message number
06	MSG_TYPE_MENU	Menu message number
07	MSG_TYPE_LED	LED message number
08	MSG_TYPE_TUNER	FM Tuner message number
09	MSG_TYPE_SOFT_TIMER	Softtimer message number
0A	MSG_TYPE_GEQ	Graphical EQ message number

Table 3. Message Definition

Length of this message including this word (a count of the 24 bit words that constitute the complete structure)
MessageDefintion (e.g., DECODER_NEXT_SONG)
Argument0
Argument...
ArgumentN

Table 4. Message Data Structure

9.2. SysPostMessage

This function is used to post module messages. The parameters are: total#ofwords,message

Messages posted from C functions use the stack and do not require special memory considerations. Some module messages are required to be sent from modules that are in assembly. These messages do not use the stack and are instead defined in memory. including a memory location for all arguments.

9.2.1. SysPostMessage Assembly Example Usage

Assembly Modules Message Definition (File: msgs.asm)

```
; Decoder Stop Message
StopMessage
                    dc      2
                    dc      DECODER_STOP
```

Message invocation (file: main.asm) requires a pointer to the beginning of the message to be passed to SysPostMessage.

```
StopDecoder
    jsr     DecoderGetStatus
    jset    #Stop,x0,_stopped

    move    #StopMessage,r0     ;Post message to stop the
decoder
    move    y:<Const_ffffff,m0
    jsr     SysPostMessage
```

9.2.2. FSysPostMessage C Example Usage

Messages are assembled on the stack and a wrapper function passes all the info in the right registers to the same SysPostMessage function that the assembly files use. (file: playerstatemachine.c)

```
SysPostMessage(2,DECODER_STOP);
```

9.3. Module Message Queue Descriptor

Word	Descriptor	Usage
0	message buffer base address	starting address of the message buffer
1	buffer modulo	modulo buffer size (size - 1), used by DSP modulo functions.
2	buffer queue size	actual size in words of the buffer
3	buffer head ptr	points to the beginning of the unread message(s)
4	buffer tail ptr	points to the end of the unread message(s)

Table 5. Module Message Queue Descriptor

The message buffer contains the actual messages sent by other modules or system routines using SysPostMessage. The message buffer size should be designed based upon the number of messages expected between module execution and the size of possible module messages.

Head and tail pointers are used to indicate new messages in the circular buffers. SysPostMessage will use the pointers to determine if there is enough room in the queue to add another message. If not the system will currently generate an unhandled error (system error - debug or reboot). The executive compares the pointers and if they are equal assumes no new messages were received.

COMPANY CONFIDENTIAL 9/26/03

9.3.1. Example Message Queue Descriptor

File: modulemsg.asm

```
DecoderQueueDescriptor
      dc      DecoderMsgQueue       ; buffer base address
      dc      MSG_QUEUE_MODULO_DECODER; buffer modulo
      dc      MSG_QUEUE_SIZE_DECODER; buffer queue size
      dc      DecoderMsgQueue ; buffer head ptr
      dc      DecoderMsgQueue ; buffer tail ptr
```

9.4. Router Table

File: modulemsg.asm.

The router table is used in SysPostMessage to determine which module message queue to load with the message. The message is routed to the associated module based upon the message id embedded in the second word of the message (Message Definition). The same MSGID is used in the router table to map the posted message to the corresponding message queue.

```
ModuleRouterTable
  dc   MSG_TYPE_DECODER|DecoderQueueDescriptor
  dc   MODULE_NUM_DECODER
  dc   MSG_TYPE_ENCODER|CaptureQueueDescriptor
  dc   MODULE_NUM_CAPTURE
  dc   MSG_TYPE_PARSER|ParserQueueDescriptor
  dc   MODULE_NUM_PARSER
  dc   MSG_TYPE_LCD|LcdQueueDescriptor
  dc   MODULE_NUM_LCD
  dc   MSG_TYPE_MIXER|MixerQueueDescriptor
  dc   MODULE_NUM_MIXER
  dc   MSG_TYPE_SYSTEM|SystemQueueDescriptor
  dc   MODULE_NUM_SYSTEM
  dc   MSG_TYPE_MENU|MenuQueueDescriptor
  dc   MODULE_NUM_MENU
  dc   MSG_TYPE_SOFT_TIMER|Fglb_SoftTimerQueueDescriptor
  dc   MODULE_NUM_SOFT_TIMER
  dc   MSG_TYPE_TUNER|TunerQueueDescriptor
  dc   MODULE_NUM_FMTUNER
  dc   MSG_TYPE_GEQ|GeqQueueDescriptor
  dc   MODULE_NUM_GEQ
```

9.5. Module Table

File: modulemsg.asm

The module table is used extensively by the executive to determine which module to load. The order of modules to execute is defined by the order of module table entry, but must also match a table of equates.

The order of module execution must match the order of the module table and these equates. The total number must be defined in MODULE_COUNT also. All defined in modulemsg.asm.

```
MODULE_NUM_DECODER      equ     0
MODULE_NUM_CAPTURE      equ     1
MODULE_NUM_PARSER       equ     2
MODULE_NUM_LCD          equ     3
MODULE_NUM_MIXER        equ     4
MODULE_NUM_SYSTEM       equ     5
MODULE_NUM_SOFT_TIMER   equ     6
```

COMPANY CONFIDENTIAL 9/26/03


```
MODULE_NUM_FMTUNER      equ     7
MODULE_NUM_GEQ           equ     8
MODULE_COUNT            equ     9 ; total # of modules
```

The control flags (Event Signal and Event Wait) used to determine if a module should be called are also in the module table. Module can define when they want to be called back (everytime, only if message received, only if timer timed out, only if button event received or any combination of these events).

Event Signal consists of control bits invoked by the operating system. Event Wait consists of control bits defined by the module. The executive runs through each module round-robin to determine if the module should be called by logical ANDing these control flags together.

If the initialization bits are set, call the initialization routine as pointed to by the module entry. If the callback is satisfied (timer, message, or button) the module processing routine is called as defined by the module table entry. The timer entries are used by the executive when a timed callback is requested.

Word	Name	Description
0	Event Signal	Bits that signal execution of the Module.
1	Event Wait	Bit mask that determines which signals the Module expects for execution.
2	Resource Number	Resource number of the code image for the given module, found in the resource.inc file.
3	Message Queue Descriptor Pointer	Points to the message queue descriptor structure for the associated module.
4	Process Routine Pointer	Pointer to the entry code for this Module.
5	Init Routine Pointer	Pointer to the init routine for this Module.
6	Signal Timer High	High word of the timer value.
7	Signal Timer Low	Low word of the timer value.
The Signal Timer parameters are used to force module execution on a timed basis. The executive will set these parameters along with the EVENT_TIMER signal in the Event Wait parameter (via a system function call). If the signal wait bit is set, the Executive will compare the Signal Timer values with the current system time and call the module if time has expired. The module must reset the time.		
8	Background Process Routine Pointer	Pointer to the code for the background process for this Module. Background tasks reside in static System memory; can perform single or multiple tasks; and may execute each time the module is executed.

Table 6. Module Table Entry Structure

COMPANY CONFIDENTIAL 9/26/03

9.5.1. Example Module Table

ModuleTable label expected by system.

```
ModuleTable
```

The decoder module conforms to the older assembly module standard, where the initialization routine is expected to be the 2nd word in the resource. The decoder initialization routine also populates the processing routine, since different decoders can be used. DecoderResourcePtr and DecoderProcessPtr are labels used by the decoder module to access the module table entry. The decoder is not automatically initialized or signaled for callback.

```
ModuleTableDecoder
    ; Decoder module
      dc  0                          ; Event signal
      dc  EVENT_INIT                 ; Event wait
DecoderResourcePtr
      dc  RSRC_DECMOD_CODE           ; resource number
      dc  DecoderQueueDescriptor     ; Message queue descriptor
DecoderProcessPtr
    dc  0                           ; Event processing routine
    dc  F_lc_u_bP_Module_overlay+1  ; initialization routine
    dc  0                           ; Signal Timer High
    dc  0                           ; Signal Timer Low
    dc  EmptyFunction               ; BackGround Processing routine
```

The softtimer module explicitly defines the initialization and processing routines. The softtimer is setup to run the initialization routine and callback upon a received message.

```
Fglb_uSoftTimerModuleEntry
    dc  EVENT_INIT                      ; Event signal
    dc  EVENT_INIT|EVENT_MESSAGE        ; Event wait
    dc  RSRC_SOFT_TIMER_MODULE_CODE     ; Resource number of code
    dc  Fglb_SoftTimerQueueDescriptor   ; Message queue descriptor
    dc  FSoftTimerModuleProcessQueue    ; Event processing routine
    dc  FSoftTimerModuleInit            ; Initialization routine
    dc  0                               ; Signal Timer High
    dc  0                               ; Signal Timer Low
    dc  EmptyFunction                   ; BackGround Processing
```

10. USER INTERFACE

10.1. Button Basics

10.1.1. Button Event Descriptions

Button configurations supported by the SigmaTel SDK software are both scan matrix (multiplexed with columns and rows) and non-scan (dedicated connections) configurations. Two files provide the setup of buttons for the system: Buttonsdef.inc and buttons.h

Buttonsdef.inc is a definition file that uses many macros to setup a configuration that the code uses to setup and monitor the corresponding GPIOs. Three types of button events are detected by routines in buttons.asm: press and hold (PH_); press and release (PR_); and any multi-button combo. Macro supporting PR, PH and 2 button event are defined.

buttons.h is a header file used by the menus to reference button events. The button event numbers used in *buttons.h* reference the button event numbers defined in *buttonsdef.inc*

10.1.2. Button Event Detection

Button sampling is performed as an interrupt using STMP hardware timer 1 (HW_TMR1). The ISR rate is dependent upon the system speed and is an entry in the speed table. When the speed changes via SysSetSpeed(), the timer counter is reinitialized using the count that represents the BUTTON_INTERVAL at the current speed.

> BUTTON_INTERVAL is the constant (defined in sysspeed.asm) representing the number of mS of the button timer interval-- default is 40mS

buttons.asm contains all the code that detects button actions via GPIO scan matrix and non-scan inputs then maps them into button event numbers.

DebounceTime is a constant (defined in buttons.asm) that is used when determining if a button has been activated.

HoldTime is a constant (defined in buttons.asm) used to determine the rate of reporting Press and Hold events (PH) -- default is 300mS. If the button is activated less than the hold time a Press and Release event (PR) will be sent to the button queue. For every "HoldTime" increment that the button is still activated a PH event will be registered.

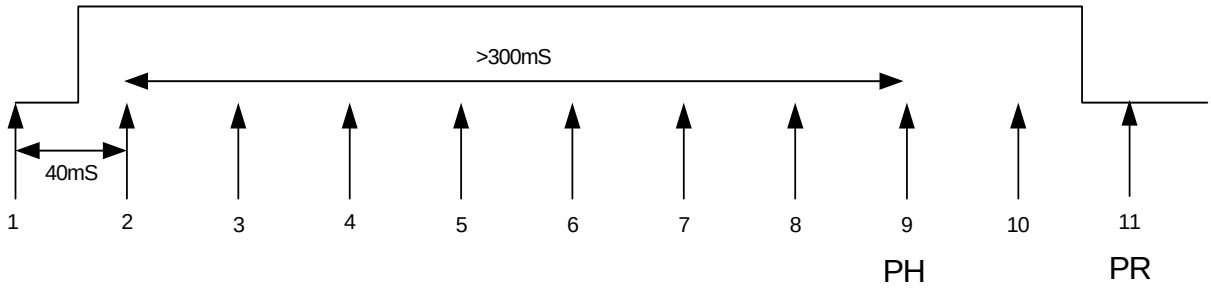


Figure 11. Example Button Event Timing Diagram

1 : button inactive - No button event in queue

2 : button changed state - STATE_DEBOUNCING

3 : button still in active state > debounce time, so button is active and moves on to the next state -- STATE_TEST_HOLD_RELEASE

4 - 8 : button still active < hold time - no event reported

COMPANY CONFIDENTIAL 9/26/03

** If the button changes state during this time, a PR event will be queued.

9 : button still active > hold time - Press and Hold event (PH) in button queue

10 : button still active < hold time - no event reported

11: button changed state to inactive - Press and Release event (PR) in button queue

** If the button is still active, a PH event will be queued every HoldTime (300mS)

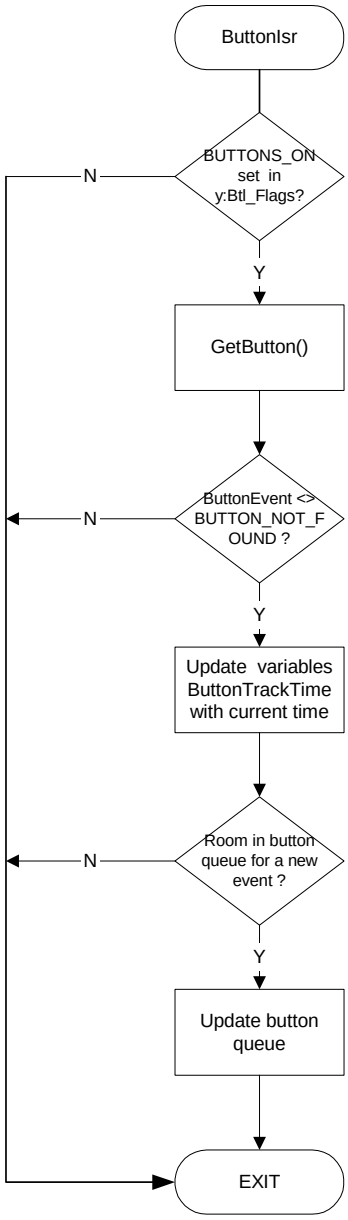


Figure 12. Button Processing Flowchart

10.1.3. Button Event Processing

The button events are queued in a circular buffer and the main loop processing signals the menu module when button events are in the button queue.

When the menu module is setup to have a button event callback, the menu module will be loaded during ExecuteModules().

The menus can use single events or combinations of button events to create various user input designs. A menu can choose to ignore a button, but the button detection routine will continue queueing button events until the buffer is full. If the button queue is full, the button detection routine not add the button events to the queue. Thus all button events detected while the button queue is full will be lost.

10.1.4. Scan Matrix Button Definitions

Refer to SigmaTel Modular Schematics on the extranet for supported scan matrix connections.

If different GPIOs are used than specified in the schematics, the scan matrix GPIOs must be changed. All rows and columns must be terminated with a pull-down resistor even if not all buttons are populated.

10.1.4.1. Scan Matrix GPIO Connections

The total number of rows and columns is defined using constants (`BUTTON_ROWS` and `BUTTON_COLUMNS`). The corresponding individual row and column pin definitions are defined using macros:

DEFINE_ROW macro RowNumber,WhichGPIO,WhichPin

DEFINE_COLUMN macro ColumnNumber,WhichGPIO,WhichPin

Column/RowNumber : user defined, sequential is more readable

WhichGPIO : the GPIO bank containing the pin***(see datasheet)***

WhichPin: the BIT representing the pin in the GPIO registers ***(see datasheet)***

EXAMPLE:

```
BUTTON_ROWS:    equ  N
DEFINE_ROW 0,GP2,1

BUTTON_COLUMNS:  equ  N
DEFINE_COLUMN 0,GP2,0
```

10.1.4.2. Scan Matrix Button Definition

The button descriptions are defined by the column and row intersection, a unique number (23 bits), and a button name. All columns and rows must have an event number mapping even if it is unused. So if 3 rows and 3 columns are defined above, COL0-COL2 and ROW0-ROW2 must be used in definition macros. Unused buttons should NOT be commented out, rather the EventNumber should be changed to $000000.

DEFINE_SCAN_BUTTON_EVENT macro Column,Row,EventNumber,ButtonName

Column: COLX where X is the matrix column number starting with 0

Row: ROWX where X is the matrix row number starting with 0

EventNumber: unique number, 23-bits; unused buttons should use $000000

ButtonName: button name that will be used when defining button events

EXAMPLE:

```
DEFINE_SCAN_BUTTON_EVENT COL0,ROW0,$000002,STOP_BUTTON
```

10.1.5. Non-Scan Button Definitions

Buttons can also be directly routed to a GPIO. Button detection is polled as described in Section 10.1.2. Button Event Detection with either button type (non-scan or scanned).

10.1.5.1. Non-scan Button Connections

The total number of non-scanned buttons is defined using constant (`BUTTON_NON_SCAN_BUTTONS`).

Non-scan buttons are defined similar to scanned buttons by specifying pin definition, and button description.

DEFINE_NON_SCAN_BUTTON macro NSNumber,WhichGPIO,WhichBit

NSNumber : user defined, sequential

WhichGPIO : the GPIO bank containing the pin***(see datasheet)***

WhichPin: the BIT representing the pin in the GPIO registers ***(see datasheet)***

EXAMPLE:

```
BUTTON_NON_SCAN_BUTTONS: equ    N
DEFINE_NON_SCAN_BUTTON 0,GP1,1
```

10.1.5.2. Non-scan Button Events

DEFINE_NON_SCAN_BUTTON_EVENT macro Number,EventNumber,ButtonName

Number: non-scan button number (0-)

EventNumber: unique number, cannot be repeated between scan and non-scan

ButtonName: button name that will be used when defining button events

EXAMPLE:

```
DEFINE_NON_SCAN_BUTTON_EVENT 0,$000004,PLAY_BUTTON
```

10.1.6. Hold Button Assignment

The Hold button is defined using label: HOLD_BUTTON_EVENT

Non-scan hold button support is built in to the button driver as a switch -- refer to the modular schematics for details. Any other hold button configuration can be handled in the menus. The gpio assignment is based upon a label referencing the non-scan button connection number NSNumber.

EXAMPLE:

```
HOLD_BUTTON_EVENT:      equ   BUTTON_EVENT_NONSCAN1
```

It is possible to disable the HOLD button functionality all together by commenting out the above line in buttonsdef.inc.

10.1.7. Button Event Assignments

Now, the button definitions must be assigned to a button event. Multiple events can be associated with each individual button. The event number is the value put into the button queue and referenced during the menus. The button event numbers in buttonsdef.inc must match the button events listed in buttons.h

The total number of button events is denoted by the constant:

```
BUTTON_MAPPED_EVENTS:   equ     N
```

10.1.7.1. Single Button Event Mapping

Single button events are defined using the following macro.

DEFINE_MAPPED_EVENT macro Number,Action,ButtonNames

Number: Event number to be mapped to specified button action

Action: Event type either PRESS_RELEASE or PRESS_HOLD

ButtonNames : name defined in button definition macro.

EXAMPLE:

```
DEFINE_MAPPED_EVENT   0,PRESS_RELEASE,PREVIOUS_BUTTON
DEFINE_MAPPED_EVENT   1,PRESS_HOLD,PREVIOUS_BUTTON
```

10.1.7.2. Double Button Event Mapping

A double button event can be defined using the following macro. Both button events for (Event1 and Event2) must have the same action type as the double button action (Action).

DEFINE_2BUTTON_MAPPED_EVENT macro Number,Action,Event1,Event2

Number: Event number to be mapped to specified button action

Action: Event type either PRESS_RELEASE or PRESS_HOLD

Event1 : single button event defined for button1

Event2 : single button event defined for button2

EXAMPLE: In this example, button event 11 will be processed when the two button associated with button events 0 & 2 are pressed and released simultaneously.

```
DEFINE_2BUTTON_MAPPED_EVENT  11, PRESS_RELEASE, 0, 2
```

10.2. Remote Control/Single Wire Button Example

Remote control refers to a single wire button detection method using the lradc. It requires a special hardware configuration. The driver is added to the SDK in the button isr, but the supplied examples do not use this feature.

Refer to the buttontest (at stmp3xxx_sdk\test\buttontest) for an example implementation.

The example provides three configuration:

1. remote control buttons are mapped onto the player key pad buttons (REMOTE1)
2. remote control buttons are mapped onto the separate buttons (REMOTE2)
3. no remote buttons (default)

If no build options are entered the test executes without remote control functionality. Use command line build options REMOTE1 or REMOTE2 to support remote button operation.

The SDK configuration detects 6 voltage levels/ranges, but only uses 5 buttons (the lowest range is ignored). This configuration corresponds with the reference schematics.

The following line in buttonsdef.inc file, defines the number of remote control buttons. The remote control sampling will not be added to the build if this label is not defined.

- define BUTTON_LRADC_BUTTONS ‘6’

The DEFINE_LRADC_BUTTON macro defines the remote control buttons.

- DEFINE_LRADC_BUTTON *buttonnumber, high level in mV*

buttonnumber = remote control button number

level in mV = the upper voltage range in mV.

COMPANY CONFIDENTIAL 9/26/03

The button must be defined such that the voltage increases with the button number.

- DEFINE_LRADC_BUTTON 0,438 ; button 0 (0-438mV)
- DEFINE_LRADC_BUTTON 1,902 ; button 1 (439-902mV)
- DEFINE_LRADC_BUTTON 2,1417 ; button 2 (903mV-1.417V)
- DEFINE_LRADC_BUTTON 3,1959 ; button 3 (1.418-1.959V)
- DEFINE_LRADC_BUTTON 4,2397 ; button 4 (1.960-2.397V)
- DEFINE_LRADC_BUTTON 5,2964 ; button 5 (2.398-2.964V)

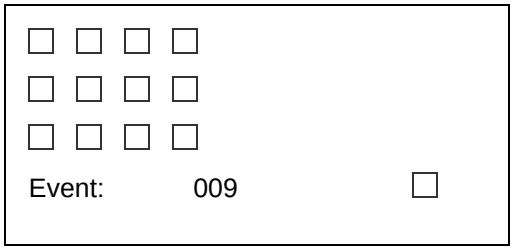
Map the remote control button number to an event number. If a remote control button requires the EXACT functionality of another button, the same EventNumber can be used and the same button event will be generated.

The REMOTE1 build option maps to the remote button to the same button event as the other buttons (i.e. remote play causes the same functionality as the PR_PLAY/PH_PLAY button events) in the file buttonsdef.inc. The line with DEFINE_LRADC_BUTTON_EVENT Number, EventNumber, ButtonName defines the relation between button and event number.

The REMOTE2 build option maps the remote button to a unique button event. In this case, a new EventNumber and button event must be defined. The additional event mapped to REMOTE_PLAY_BUTTON (and added to buttons.h) will not be used unless the PLAY_BUTTON EventNumber is a unique number.

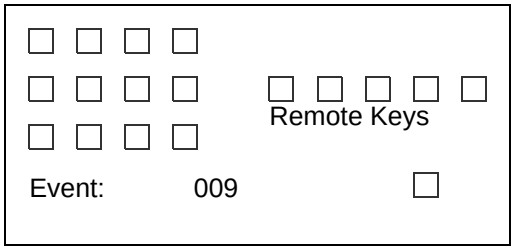
To modify button event, change the following descriptions:

- Add button event mapping----DEFINE_MAPPED_EVENT .
- Update the total number of mapped events BUTTON_MAPPED_EVENTS. Update the ‘buttons.h file to match the definitions in the buttonsdef.inc file.



Display screen for REMOTE1 build option (map remote control keys onto keypad keys).

Figure 13.



Display screen for REMOTE2 build option (map remote control keys onto separate keys).

Figure 14.

10.3. Jog Dial Example

A jog dial, as shown in the reference schematics, could be used to reduce the size of the player. The jog dial is connected to the rows and columns of the scan matrix and acts like any other scan button event. There are 3 total positions and 6 events (3 positions plus press release/hold).

Since the total numbers of buttons has been reduced, it will be desirable to perform multiple functions with the same button events. For instance, the left/right moves could be used to change the volume setting or to scan across songs. The following example shows how to implement this feature while in the Music Menu.

10.3.1. Jog Dial Buttons Events

- PR_LEFT – Press and release to the left position
- PH_LEFT – Hold in the left position
- PR_RIGHT – Press and release to the right position
- PH_RIGHT – Hold in the right position
- PR_MIDDLE – Press and release to the middle position
- PH_MIDDLE – Hold in the middle position

10.3.2. Jog Dial Code Modifications

The middle button can be used to toggle between scan mode (REW/FF) and volume update. These modes are then used to determine how the left and right buttons control the system.

- JogButtonMode – Stores current button mode which is used to determine functionality
- SCAN_MODE – define used to set scan mode.
- VOL_MODE – define used to set volume mode.

```
case PR_MIDDLE://change button mode
if (iJogButtonMode == SCAN_MODE)
      iJogButtonMode = VOL_MODE;
else
      iJogButtonMode = SCAN_MODE;
break;

case PH_MIDDLE:
//use as entry to main menu for access to settings, voice,
EQ, etc.
break;

case PR_LEFT: (or PR_RIGHT)
  if(iJogButtonMode == SCAN_MODE)
      //Process as PR_FF or PR_REW
  else //VOL_MODE
      //Process as PR_VOL_UP or PR_VOL_DOWN
  break;

case PH_LEFT: (or PH_RIGHT)
  if(iJogButtonMode == SCAN_MODE)
      //Process as PH_FF or PH_REW
  else //VOL_MODE
      //Process as PH_VOL_UP or PH_VOL_DOWN
  break;
```

10.4. Rotary encoder

The SDK only supports the two-phase (A, B) incremental rotary encoder. The two-phase signals are phase differentiated, and on each edge, an up-count is performed if A phase is leading, or a down count is performed if B phase is leading. The firmware provides an absolute position of the encoder within the specified range.

Many mechanical encoders has detents or clicks with n-pulse per detent or click. The firmware automatically counts the number of clicks if the number of phase per detent is correctly specified.

10.4.1. Device driver code

The device driver is GPIO isr based using either the edge sensitive or level sensitive interrupt for the best performance. This driver will capture the encoder rotation and translate it into the direction and normalized value between the defined maximum and minimum values (g_wMaxEncoderValue and g_wMinEncoderValue).

The variable bEncoderRotated is set TRUE during the isr.

The source code for this device is '\DeviceDriver\UserInput\RotaryEncoder\rotaryencoder.c'. An include file '\inc\rotaryencoder.h' declares the 3 global variables and the function prototypes for this driver.

10.4.2. Encoder absolute position

The rotary encoder device driver reports the current absolute position of the encoder in a global variable, g_wRotaryEncoderValue, which must be initialized before using it. The range is defined by another two global variables, g_wMaxEncoderValue and g_wMinEncoderValue. These 3 variables are initialized to these default values, INIT_ENCODER_VALUE, MAX_ENCODER_VALUE, and MIN_ENCODER_VALUE, respectively. The absolute position can represent different units as defined by the specified range. Example ranges include: 0 - 100 % volume level, or the 88 - 108 MHz FM radio frequencies, or the number of music tracks.

10.4.3. Encoder Initialization

This driver is initialized by calling RotaryEncoderInit() right after the call to StartProject() in the Exec() routine in the main.asm. The 3 global variables will be initialized in this routine.

10.4.3.1. GPIO pins selection

The encoder is connected to the STMP3xxx through two GPIO pins. The default pins, HW_ROTARY_A and HW_ROTARY_B, are defined in the source code. The signals are negative true.

10.4.3.2. Define the number of phases per detents

The default value is defined as 'PHASE_PER_DETENT' = 4 in rotaryencoder.c

10.4.4. Encoder Application Interface

Two functions RotaryEncoderNotifySetup() and HandleRotaryEncoder() are available to post messages based on the encoder position. These functions both rely on a global variable modified by the device driver isr. Care must be taken if both functions are used because the variable bEncoderRotated is set FALSE when a message is posted by either function.

10.4.4.1. UI Callback Mechanism

The routine RotaryEncoderNotifySetup() posts the input module message when the encoder value changes from the passed in value. This is done so the encoder can notify the menu system about the change. If the value passed to this routine is different from the value of the current encoder value, the message is immediately posted.

Each time the UI receives the message that the encoder value has changed, a new call to this routine should be made to detect the next change in encoder position.

10.4.4.2. Encoder Position Change Notification

By calling HandleRotaryEncoder() routine inside the Exec(), any UI message can be posted when the encoder position is changed. This function does not compare against an input value but will post a message based on the state of the variable bEncoderRotated.

10.4.5. Test Examples

The SDK provides two tests that can be used to check the operations of the encoder. One example, QEtest, is the modified button test to include the reporting of the encoder position between 0 and 100. This can be used to test the hardware. The other test, fmtunertest, requires an fm tuner module where the rotary encoder is used as the tuning wheel for the fmtuner.

10.4.6. Application Example

Refer to the extranet for an application note that uses the rotary encoder as the input mechanism in the lcdexample, based on SDK 2.320.

10.5. Graphical LCD

The graphical LCD messaging module is based upon the Epson SED1565 controller. Many controllers are compatible with it. Compare the messages and hardware interface with the Epson LCD datasheet (provided on the SigmaTel Extranet) to determine compatibility. Differences in glass size and resolution can be accounted for in header files (project.inc and display.h).

10.5.1. LCD Initialization

Configuration registers initialized at startup are stored in a resource file (system_lcd_init_seq.src). The table below shows the organization of the resource file. The resource file may be changed to update these initial register values to adjust the configuration of the LCD for a given hardware design or be customized for a given display hardware type.

COMPANY CONFIDENTIAL 9/26/03

This resource file used during LCD module initialization in the SDK examples use the following commands.

LcdInitSeq Resource Data Structure Contents	Number of DSP words
Number of words	1 DSP Word
Power Control Register.	1 DSP Word. Data in 8 LSBs.
Display Line.	1 DSP Word. Data in 8 LSBs.
Display Mode.	1 DSP Word. Data in 8 LSBs.
Segment Display Mode.	1 DSP Word. Data in 8 LSBs.
Internal Regulator Resistor Ratio.	1 DSP Word. Data in 8 LSBs.
Segment Output.	1 DSP Word. Data in 8 LSBs.
Volume Mode.	1 DSP Word. Data in 8 LSBs.
Volume Register (contrast)	1 DSP Word. Data in 8 LSBs.
Display On/Off	1 DSP Word. Data in 8 LSBs.
Display Data.	1 DSP Word. Data in 8 LSBs.

Table 7. SYSTEM_LCD_INIT_SEQ RESOURCE

10.5.2. Display Resources

In SDK examples, most display items are resources stored on NAND in the resource.bin. When a print resource message is received, the LCD module will load the resource into RAM (UserScratchY) then update the display using the resource information at the given coordinates.

A utility is provided in the SDK (buildres.exe) that will convert bitmaps and strings to resource files. The lcdexample in the SDK provides a makefile that will automatically update the resources as the source files change. The resources can also be created manually using buildres.exe in a command window.

10.5.3. Bitmaps

Bitmaps must be monochrome (black and white only), and each pixel maps to a bit sent to the LCD. The buildres.exe utility converts .bmp files to resource files. The controller interprets the incoming data as vertical pixels, so the .bmps are converted to resource files such that a resource byte is mapped to LCD pixels as follows:

```
$RESOURCE_TYPE
BITMAP
$LABEL
play_icon_with_border
$REC_SIZE
9                       <---- # words of resource (decimal)
$DATA
000012                  <---- # data bytes
000003                  <---- Resource type (bitmap)
00000c                  <---- Bitmap width
000008                  <---- Bitmap height
8181ff                  <---- Data (see below for decode)
bf8181
81899d
ff8181
ONLY DATA IN BOLD/BLUE GETS LOADED INTO MEMORY.
```

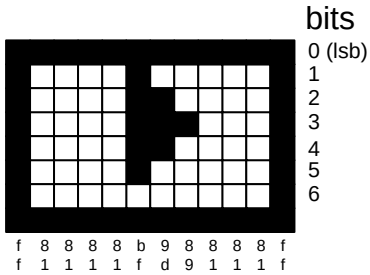


Figure 15.

10.5.4. Strings

Text strings can also be converted to resources using buildres.exe. The text string is interpreted as DBCS and mapped to the font set used by the player.

Refer to the lcdexample menu options in the lcd for illustration (i.e. string_music_menu.src).

10.5.5. Fonts

String resources and internal strings map DBCS codepages to the font set assigned in the player. Current SDK implementation limits the font set to a single codepage.

Refer to Appendix APPENDIX E. Building Custom Font Resources for more specific information on creating custom fonts.

10.6. Displaying Strings

Internal strings are interpreted as DBCS characters and mapped to the font set used by the player.

In these SDKs, ID3 tags are assumed to DBCS. Support for Unicode ID3 tags will be available in SDK2.2xx release.

Directory separators in filename paths are forward slashes (/) instead of back slashes to accommodate some DBCS code pages that use back slashes to denote a trailing character.

10.7. Contrast

Default contrast settings in the player are defined in project.inc. The contrast is a saved setting in the player firmware and will be adjusted during the lcd module initialization. The usbmsc firmware does not save settings, so the default value in project.inc is always used.

The actual displayed contrast for a given value is dependent upon the voltage applied to the LCD (typically Vddio).

It is recommended to adjust the range of contrast values during customizations so that the player will not save a contrast setting that prevents the user from seeing the display (too dark or light to read). Refer to project.inc for instructions on how to adjust the contrast range values (LCD_MAX_CNTRST and LCD_MIN_CNTRST) for the voltages on the player design.

10.8. Backlight

A backlight build option is available for the SDK2.320 release. When the option is enabled in the make file, a backlight GPIO and softtimer will be implemented. Within each menu, a button press (EVENT_BUTTON) will turn on the GPIO for the backlight and post a softtimer message to turn off. Since all messages are sent the playerstatemachine or the

COMPANY CONFIDENTIAL 9/26/03

recorderstatemachine, the backlight off message (MENU_MSG_TURN_OFF_BACKLIGHT) will be handled there instead of each individual menu.

The backlight define statements can be modified to assign the GPIO and to define the amount of time the backlight should be lit. A new Backlight Menu was also added to the settings menu so the user could disable the backlight if desired.

For the USBMSC firmware, the option is also available in their respective make files. All code resides within the LCD routine. The backlight will turn on during start-up, when reading or writing and will shut down when idle.

10.9. Graphical LCD Messaging Module API

Message Name	Arguments	Message Description
LCD_CLEAR_RANGE	X coordinate in pixels Y coordinate in pixels M pixels in X direction N pixels in Y directions	Clears NumChars starting at the (x,y) coordinate for a range of (m,n) This message does not affect the contrast.
LCD_PRINT_RANGE _RSRC	X coordinate in pixels Y coordinate in pixels BitmapResourceNumber	This message will cause the LcdModule to display the bitmap indicated by the BitmapResourceNumber argument starting at the location specified by the (x,y) coordinate.
LCD_PRINT_RANGE_INV_RSRC	X coordinate in pixels Y coordinate in pixels BitmapResourceNumber	Identical to LCD_PRINT_RANGE_RSRC, except output is displayed inverted.
LCD_PRINT_RANGE _ADDR	X coordinate in pixels Y coordinate in pixels Address of bitmap	This message will cause the LcdModule to display the bitmap stored at the Address argument starting at the location specified by (x,y) coordinate. The bitmap data is expected to be stored in Y memory.
LCD_PRINT_RANGE_INV_ADDR	X coordinate in pixels Y coordinate in pixels Address of bitmap	Same as LCD_PRINT_RANGE_ADDR but the data is inverted before sending it.
LCD_PRINT_STRING _RSRC	X coordinate in pixels Y coordinate in pixels StringResourceNumber	This message will cause the LcdModule to display the string indicated by the StringResourceNumber argument starting at the location specified by the (x,y) coordinate.
LCD_PRINT_STRING _INV_RSRC	X coordinate in pixels Y coordinate in pixels StringResourceNumber	This message will cause the LcdModule to display the string indicated by the StringResourceNumber argument starting at the location specified by the (x,y) coordinate.

Table 8. Graphical LCD Messaging Module API

COMPANY CONFIDENTIAL 9/26/03

LCD_PRINT_STRING _ADDR	X coordinate in pixels Y coordinate in pixels Address of string	This message will cause the LcdModule to display the string indicated by the Address argument starting at the location specified by the (x,y) coordinate. The string is expected to be stored in Y memory.
LCD_PRINT_STRING _INV_ADDR	X coordinate in pixels Y coordinate in pixels Address of string	Identical to LCD_PRINT_STRING_ADDR, except the display is inverted.
LCD_SCROLL _DISPLAY	-	Unsupported
LCD_SET_CONTRAST	Contrast % value	Sets the contrast to a particular percentage value (0-100%).
LCD_INC_CONTRAST	None	Increases the contrast by 2%
LCD_DEC_CONTRAST	None	Decreases the contrast by 2%
LCD_SET_FONT	Font index	This message specifies which font to use when displaying strings and numbers. The fonts are defined in the font table, and the default font index is 0. Current SDKs only allow different fonts for the same codepage (i.e. bold, italic,etc) but not different code pages. Future SDKs will expand this functionality to different codepages also.
LCD_PRINT_NUMBER	X coordinate in pixels Y coordinate in pixels Number Field width Leading character	This message displays a number at the specified (x,y) coordinate. The field width specifies how many characters to use to display the number. If the number is smaller than the field width, then the field is left filled with leading characters. A leading character is usually an ASCII space character or an ASCII zero character.
LCD_PRINT_NUMBER_INV	X coordinate in pixels Y coordinate in pixels Number Field width Leading character	Identical to LCD_PRINT_NUMBER, except output is displayed inverted.
LCD_PRINT_TIME	X coordinate in pixels Y coordinate in pixels Minutes Seconds	This message displays a time value at the specified (x,y) coordinate. The format of the time is MM:SS where MM is minutes and SS is seconds.

Table 8. Graphical LCD Messaging Module API (Continued)

COMPANY CONFIDENTIAL 9/26/03

LCD_PRINT_TIME_INV	X coordinate in pixels Y coordinate in pixels Minutes Seconds	Identical to LCD_PRINT_TIME, except the output is displayed inverted
LCD_PRINT_TIME_LONG	X coordinate in pixels Y coordinate in pixels Hours Minutes Seconds	This message displays a time value at the specified (x,y) coordinate. The format of the time is HH:MM:SS where HH is hours, MM is minutes, and SS is seconds.
LCD_PRINT_TIME_LONG_INV	X coordinate in pixels Y coordinate in pixels Hours Minutes Seconds	Identical to LCD_PRINT_TIME_LONG, except the output is inverted.
LCD_PRINT_STRING_UNICODE_ADDR	X coordinate in pixels Y coordinate in pixels Address of unicode string	Displays the unicode string pointed to by address at location (x,y)
LCD_PRINT_STRING_UNICODE_INV_ADDR	X coordinate in pixels Y coordinate in pixels Address of unicode string	Same as LCD_PRINT_STRING_UNICODE_ADDR except the output is inverted.
LCD_PRINT_STRING_UNICODE_RSRC	X coordinate in pixels Y coordinate in pixels Resource of unicode string	Displays the unicode string stored in the resource at location (x,y)
LCD_PRINT_STRING_UNICODE_INV_RSRC	X coordinate in pixels Y coordinate in pixels Resource of unicode string	Same as LCD_PRINT_STRING_UNICODE_RSRC, except the output is inverted.
LCD_PRINT_UNICODE_CHAR	X coordinate in pixels Y coordinate in pixels Unicode Char #	Displays a single unicode character at location (x,y)
LCD_PRINT_UNICODE_CHAR_INV	X coordinate in pixels Y coordinate in pixels Unicode char #	Identical to LCD_PRINT_UNICODE_CHAR, except the output is inverted
LCD_BEGIN_FRAME	-	Informs the LCD module that the frame update is beginning and that the LCD should not be updated until the frame is completed.
LCD_END_FRAME	-	Allows the LCD module/driver to update the physical LCD device.

Table 8. Graphical LCD Messaging Module API (Continued)

COMPANY CONFIDENTIAL 9/26/03

LCD_SET_FRAMEBUFFER	Address of new frame-buffer bitmap (or NULL)	Sets the target frame buffer to be a bitmap in memory pointed to by address passed in. If NULL is passed in, then the framebuffer is returned to the default framebuffer. See below for more information.
LCD_PUSH_MASK	X coordinate in pixels Y coordinate in pixels M pixels in X direction N pixels in Y directions	Adds a new masking area to the mask stack. The display driver should not draw outside of the union of all masks on the stack.
LCD_POP_MASK	-	Pops the latest mask off the stack

Table 8. Graphical LCD Messaging Module API (Continued)

10.9.1. Framebuffer and LCD

The driver model for SDK 2.2xx has changed with respect to the LCD. The driver has been split into three layers:

Module Consumes messages and converts them to function calls that are generic and abstract. This layer does not depend on any particular LCD display controller, or method of drawing to the screen.

SAL Provides a common interface for the module to draw to the screen.

HAL Abstracts the physical interface of the LCD controller for the SAL.

Each layer can handle messages, or pass them down to a lower level if necessary. For example, LCD_SET_CONTRAST is passed through each layer and is processed in the HAL.

10.9.1.1. Module

The Display Module is meant to be a high level display library that handles the common tasks of drawing bitmaps, strings, etc. It will load up the bitmaps, or convert strings as necessary to bitmaps to pass down to the SAL for it to copy the bitmaps to the display device.

New features and/or messages will be processed here.

10.9.1.2. Software Abstraction Layer (SAL)

The SAL is tuned toward abstracting the logical aspects of the LCD controller into software on the DSP. The two options available currently are the framebuffer SAL and the direct-to-LCD SAL.

The framebuffer SAL has an internal framebuffer that it draws to, and then sends the updated framebuffer to the LCD controller chip. The benefits of this implementation of the SAL are bitmaps of any size drawn at any location on the screen, plus deferring updating the display until all updates are complete to prevent flickering.

The direct-to-LCD SAL draws the bitmaps directly to the LCD controller. It requires bitmaps be drawn at a 8 pixel vertical boundary, though they are necessarily required to be a multiple of 8 pixels tall. There are no horizontal size or placement requirements. This SAL is less processor intensive when not scrolling and has a smaller memory footprint than the framebuffer SAL.

10.9.1.3. Hardware Abstraction Layer (HAL)

The HAL abstracts the physical interface of communicating with the LCD hardware. If the communication with the LCD controller is different than that of the reference design, or the pins need to change for a reason, this is the section of code to change.

10.9.1.4. Messages

The messages are generally consumed by the module, but some messages which are not handled by the module are passed down to the SAL, and then to the HAL if not handled there.

10.9.1.5. Framebuffer Details

The framebuffer implementation has specific messages meant for it: LCD_SET_FRAMEBUFFER, LCD_BEGIN_FRAME, LCD_END_FRAME.

Usually, the LCD module will copy the framebuffer to the LCD controller when it has finished processing all of its messages. LCD_BEGIN_FRAME and LCD_END_FRAME are to be used around an update to the display to prevent this. The LCD_BEGIN_FRAME message will tell the display module/driver to not output the updated framebuffer to the LCD controller until LCD_END_FRAME is received. This allows the menu module to fill up the LCD message queue and allow it to empty several times without letting the interim results be displayed which could cause flickering on the LCD if not prevented.

The framebuffer is a region in memory which is formatted exactly like a bitmap. The default framebuffer is called the root framebuffer. This root framebuffer is the only one that will be copied to the LCD controller in the current framebuffer driver. The developer can, however, change the render target to generate bitmaps for later use. For example, one optimization to help scrolling speeds is to render the string into a bitmap, then scroll that bitmap across the screen. This improves performance by removing NAND accesses when writing the string (each character of a string involves 4-7 NAND accesses). The steps involved in this would be as follows:

1. Compile with a buffer large enough for whatever string expected: UINT BitmapBuffer [2+(((X_SIZE*Y_SIZE)+2)/3)];
2. Determine actual width of string GetTextWidthAddressUnicode(&(StringAddress[0]))
3. Set the X and Y size (first two words of a bitmap are the X and Y size)
4. SysPostMessage(3,LCD_SET_FRAMEBUFFER, &(BitmapBuffer[0]))
5. SysPostMessage(5, LCD_PRINT_STRING_UNICODE _ADDR,0,0,&(StringAddress[0]));
6. SysPostMessage(3,LCD_SET_FRAMEBUFFER,NULL)

To use this new bitmap of the string:

```
SysPostMessage(5,LCD_PRINT_RANGE_ADDR,x_pos,y_pos,&(BitmapBuffer[0]));
```

10.9.2. Resize the LCD Display

Resizing the LCD display involves two parts, USB LCD interface and player LCD interface, and code changes should be done at both module level and HAL (device driver) level. In this document, discuss is limited at module level.

All constants, files, and examples are based on SDK 2.3xx below.

10.9.2.1. Changes for USB LCD interface

- Values change for the constants in ***\stmp3xxx_sdk\projects\sdk\lcdexample\dcc\dcclcd.asm***:
  - LOC_Y_TOP equ 8 ;The most top row of text display
  - LOC_X_LEFT equ 0 ;The most left column of display
  - LOC_Y_TEXT equ 48 ;The start row of display text READY
  - LOC_Y_NOTES equ 24 ;The start row of display Notes
  - LOC_X_MAX_LEFT equ 35 ;The most left position Notes can go
  - LOC_X_MAX_RIGHT equ 70 ;The most right position Notes can go
  - LOC_Y_VERSION equ 8 ;The start row of display version
  - LOC_X_VERSION equ 32 ;The start column of display version
- The X/Y coordinates of the lcd messages may also need to be changed in:
  - DisplayConnectRight
  - ClearText
  - ClearDisplay
- Resize bitmap files in ***\stmp3xxx_sdk\projects\sdk\lcdexample\dcc\rsrc***
- Rebuild dcc and usbmsc module using the ALL build option

10.9.2.2. Changes for player LCD interface

- Change values for the constants in: ***\stmp3xxx_sdk\projects\sdk\lcdexample\player\display\display.h***
  - LCD_SIZE_X 98;the number of pixels in X direction.
  - LCD_SIZE_Y 64; the number of pixels in Y direction.
  - LCD_SIZE_ROW 8; the number of row of text will be displayed.
  - SCREEN_WIDTH 98; the width of the LCD screen.
  - SCREEN_HEIGHT 64; the height of the LCD screen.
- Resizing The logo bitmap file st_bw1.bmp in C:\stmp3xxx_sdk2311\bitmaps\logos\ directory.
- The X/Y coordinates and ranges of the lcd messages may also need to be changed for the project.
- ***\stmp3xxx_sdk\projects\sdk\lcdexample\player\rsrc\bitmaps***
- Update X/Y coordinates of LCD messages in the player directory.
- Rebuild the player using the ALL build option.

10.9.2.3. The speed of display on LCD

- For displaying one character(8x8) on LCD will take 5.3ms including loading time 1.15ms.
- For transmitting whole buffer(96x64) onto LCD will take 6.9ms.

10.9.3. LED GPIO assignment

ledtable.asm contains a table that provides GPIO information for LEDs. Use the file provided with ledexample in the SDK as a template because certain labels are required as well as order of table entries. The LedTable is required to reside in Y memory.

Required labels:

- LedTable: label indicating start of table -- exact name expected
- LED_TABLE_LENGTH: number of items in each table entry -- fixed value do not change. Code provided uses math
- LED_NUMBER: equate representing total number of leds supported. There must be a table entry for each led.

10.9.4. LedTable entry

Each led to be controlled by the led module must have a 9 word table entry, using the order defined in the example.

The label is not used in module, but provided for readability.

The first 3 words are the GPIO register names used to control each LED. The register names are defined in regs3410.h, but only the number needs to be changed in the examples to the corresponding GPIO bank. The order should not be changed. Determine which pin or GPIO number the LED to control is connected to on the STMP3xxx device.

Refer to the D-Major Audio Decoder datasheet to determine which GPIO bank the pin is in.

The next two words define the actual bit position in the registers, providing an ON and OFF bit mask. The bit position for a given GPIO will be the same for all three registers. The ON mask is a hex value with a single bit set, this represents the GPIO position in the hardware registers. The OFF mask is a hex value with all bits set except the bit representing the GPIO position in the hardware register. The OFF/ON masks are inverses of each other.

The last 4 words are used by the led module and should be initialized to 0.

```
Led0        dc      HW_GP0ENR     ; GPIO Bank 0 Enable Reg-
ister
            dc      HW_GP0DOER    ; GPIO Bank 0 Direction
Register
            dc      HW_GP0DOR     ; GPIO Bank 0 Data Register
            dc      $000020       ; ON mask to set GPIO (OR)
            dc      $FFFFdf       ; OFF mask to clear GPIO
(AND)
            dc      0             ; On Time
            dc      0             ; Off Time
            dc      0             ; current on time
            dc      0             ; current off time
```

10.9.5. LED Module

LED control is provided through a messaging module. The module will control blinking the LEDs at a specified dutycycle or a constant ON/OFF state. The module abstracts the control from the hardware level and provides the timing control required when blinking the LED.

As with all modules, if you want to add led support the led module must be entered into the module table. Refer to ledexample provided in the SDK for reference on setting up the led module in the module table and declaring the message queue.

COMPANY CONFIDENTIAL 9/26/03

10.9.6. LED initialization

During module initialization, the GPIOs are enabled and setup to be outputs with LEDs in the off state.

10.9.7. LED Module Messaging API

Three messages are supported in the LED module. The LED number refers to the order of the LedTable entries. The first entry is 0, and each entry after that is incremented by one. In the example LedTable, the labels “Led0”, “Led1”, etc. denote the LED number to use.

LED_RESET: clears the last 4 words of each table entry to reset the internal timing variables.

LED_CONTROL: Using OnTime and OffTime parameters, the module will turn the specified led off or on or blink it. Refer to Table 9 LED Messagesfor states. If a non-zero value is provided for the OnTime and OffTime, the LED module will blink the LED at the specified rate until another message is send turning the LED off.

LED_DLY_ON_CONTROL: This message with the delay on time set to 0 functions like LED_CONTROL message. The delay on time allows the specified LED to come on later so many LEDs can be grouped together to flash in a controlled manner.

# words	Message Name	Arguments	Message Description
5	LED_CONTROL	LED Number OnTime OffTime	Controls the state of an LED: On Time Off Time Action 0 x LED Off >0 0 LED On y z LED on for y ms and of for z ms
6	LED_DLY_ON_CONTROL	LED Number OnTime OffTime DelayOnTime	Controls the state of an LED: On Time Off Time Action 0 x LED Off >0 0 LED On y z LED on for y ms and of for z ms DelayOnTime turns the LED on after the specified delay in ms
3	LED_RESET	LED Number	Forces the LED to the default state (on or off).

Table 9. LED Messages

10.9.8. Example LED Messages

```
SysPostMessage(5,LED_CONTROL,iLed,50,450);
```

where, `iLed` is a variable that gets assigned prior to call indicating which LED (between 0 and LED_NUMBER-1). After the LED module processes this message, the specified LED will be turned on for 450mS the off for 50 mS continually until another message affecting this same LED number is processed.

```
SysPostMessage(5,LED_CONTROL,PLAY_STATE_LED,1,0);
```

where, PLAY_STATE_LED is a #define representing the LED number (between 0 and LED_NUMBER-1). After the LED module processes this message, the specified LED will be turned on continually until another message affecting this same LED number is processed.

COMPANY CONFIDENTIAL 9/26/03


```
SysPostMessage(5,LED_CONTROL,1,0,1);
```

After the LED module processes this message, Led1 or the LED corresponding to the second LedTable entry will be turned off until another message controlling Led1 is processed.

10.9.9. Example LED flashing Messages

These examples use 5 LEDs. They are demonstrated in the Seqledtest where the on time is LED_SEQ_ON_TIME= 100 ms and the off time for each LED is LED_SEQ_OFF_TIME = LED_SEQ_ON_TIME * (LED_NUMBER-1) where LED_NUMBER = 5. The delay times are defined here:

#define LED_DLY_TIME1LED_SEQ_ON_TIME

#define LED_DLY_TIME2LED_SEQ_ON_TIME+LED_DLY_TIME1

#define LED_DLY_TIME3LED_SEQ_ON_TIME+LED_DLY_TIME2

#define LED_DLY_TIME4LED_SEQ_ON_TIME+LED_DLY_TIME3

10.9.9.1. Right to left flashing example

```
SysPostMessage(6,LED_DLY_ON_CONTROL,0,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,0);   // Instant on
SysPostMessage(6,LED_DLY_ON_CONTROL,1,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME1);
SysPostMessage(6,LED_DLY_ON_CONTROL,2,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME2);
SysPostMessage(6,LED_DLY_ON_CONTROL,3,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME3);
SysPostMessage(6,LED_DLY_ON_CONTROL,4,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME4);
```

10.9.9.2. Left to right flashing example

```
SysPostMessage(6,LED_DLY_ON_CONTROL,4,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,0);   // Instant on
SysPostMessage(6,LED_DLY_ON_CONTROL,3,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME1);
SysPostMessage(6,LED_DLY_ON_CONTROL,2,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME2);
SysPostMessage(6,LED_DLY_ON_CONTROL,1,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME3);
SysPostMessage(6,LED_DLY_ON_CONTROL,0,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,LED_DLY_TIME4);
```

10.9.9.3. Moving out flashing example

```
SysPostMessage(6,LED_DLY_ON_CONTROL,0,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,200);
SysPostMessage(6,LED_DLY_ON_CONTROL,1,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,100);
SysPostMessage(6,LED_DLY_ON_CONTROL,2,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,0);   // instant on
SysPostMessage(6,LED_DLY_ON_CONTROL,3,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,100);
SysPostMessage(6,LED_DLY_ON_CONTROL,4,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,200);
```

10.9.9.4. Moving in flashing example

```
SysPostMessage(6,LED_DLY_ON_CONTROL,0,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,0);   // Instant on
SysPostMessage(6,LED_DLY_ON_CONTROL,1,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,100);
SysPostMessage(6,LED_DLY_ON_CONTROL,2,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,200);
SysPostMessage(6,LED_DLY_ON_CONTROL,3,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,100);
SysPostMessage(6,LED_DLY_ON_CONTROL,4,LED_SEQ_ON_TIME,LED_SEQ_OFF_TIME,0); // Instant on
```

11. FEATURE SET

The feature set of the player is a list of functions which the SDK player software is capable of supporting.

11.1. Track Playback Operation

To implement track playback and all of the functionality associated with it, certain procedures must be adhered to. The different functions of track playback are listed along with the necessary calls to the PlayList API and the PlayerLib.

11.1.1. Play/Pause

When the PLAY button is pressed, the PlayerLib command PlayerLib_GetCurrentSong() is called to set up the current song to be played, it also loads up the correct decoder type depending upon the file extension. The decoder state can be set to DECODER_STATE_PLAY using the PlayerLib command PlayerLib_SetState ().

If the decoder is currently stopped the decoder will begin track playback. The volume level will be faded-in from the mute state using the mixer master volume.

If the decoder is not in the stopped mode then the decoder state will be set to DECODER_STATE_TOGGLE, which will cause the playback to pause if playing or it will cause the playback to start if paused.

The volume level will be faded-out to mute using the mixer master volume when going from the playing state to either pause or stop.

Bits in *Fg_wDecoderSR* represent the state of the decoder. See Section 17.3.4. DecoderSR / DecoderCSR Flag Definition.

The MP3 and WMA decoders will automatically send a parser message, which will be intercepted and routed to the menu manager, to get the next track and continue playing when the end of the track or if an invalid track is reached. Operation when reach the end of the playlist is dependent upon the selected playmode.

The ADPCM decoder will stop on the current track when the end of the track is reached.

11.1.2. Stop

When the STOP button is pressed or the end of the playlist is reached the PlayerLib_SetState() command is used to set the state of the decoder to the DECODER_STATE_STOP state, the decoder will then stop decoding and fade out to muted state using the mixer master volume. If the STOP button is pressed a second time the Playlist_SetPlaySet will reset the current list to the beginning of the list, it will also reshuffle if shuffle is on.

11.1.3. FF

The FF seek function (action typically associated with press and hold FF button) is activated with the PlayerLib_FastForward() PlayerLib command. The decoder will advance X number of bytes. If the EOF is less than the number of bytes to skip, the next track is loaded and the track is advanced by the remaining number of bytes to skip. Track access wraps around when the end of the playlist is reached. If PH_FF through the last track of the playlist the first track will be loaded and FF seeking continues.

The FF track skip function (action typically associated with press and release of the FF button) is activated by the QuickSwitchSong() function, which Increments/decrements the track number and sets up a timer to tell the player state machine to change tracks. The QuickSwitchSong() function in turn calls Playlist_GetNextSongFileInfo() or Playlist_GetPreviousSongFileInfo() to skip to the next song.

COMPANY CONFIDENTIAL 9/26/03

11.1.4. RW

The RW seek function (action typically associated with press and hold RW button) is activated with the PlayerLib_Rewind() PlayerLib command. The decoder will go back X number of bytes. If the beginning of file is less than the number of bytes to skip, the previous track is loaded and the track is rewind by the remaining number of bytes to skip. Track access wraps around when the beginning of the playlist is reached. If PH_RW through the

The RW track skip function (action typically associated with press and release of the RW button) is activated by the QuickSwitchSong() function, which Increments/decrements the track number and sets up a timer to tell the player state machine to change tracks. The QuickSwitchSong() function in turn calls Playlist_GetNextSongFileInfo() or Playlist_GetPreviousSongFileInfo() to skip to the next song.

11.1.5. Variable FFWD & RWND Scan Speed

The scanning of songs is a fairly simple process; a message is sent to the decoder from the playerstatemachine menu and the decoder processes the message by skipping ahead a set number of bytes inside the file. Two separate variables control the amount of bytes to skip depending upon the decoder loaded:

Mp3DecoderFfwdRwndValue

WmaDecoderFfwdRwndValue

These variables are initialized to the FFWD_RWND_BYTE_SKIP value, which is calculated to be:

$$\text{FFWD\_RWND\_ BYTE\_SKIP} = @\text{CVI}\left\{3@\text{FLR}\left[t_{sec}(128\ \text{kbps})\left(\frac{1byte}{8bits}\right)\left(\frac{1}{3}\right)\right]\right\}$$

$$\propto$$

$$\text{FFWD\_RWND\_ BYTE\_SKIP} \quad \text{equ} \quad (@\text{CVI}(3 * (@\text{FLR}((\text{SEC} * 128000/8)/ 3))))$$

Figure 16.

- **@FLR**, the floor function, also called the greatest integer function, gives the largest integer less than or equal to the input.
- **@CVI** converts the result of expression to an integer value
- The three makes sure the value is divisible by 3 to go from bytes to words
- t_{sec} is the desired time interval between skips, this should be the displayed time interval

The desired number of seconds to be skipped per one **FFWD** or **RWD** decoder message is configured in two places. **SEC** is the initial value used to calculate the default

FFWD_RWND_BYTE_SKIP that the above formula uses as t_{sec}; **SECONDS_TO_SKIP** is the value that is used to dynamically calculate the byte amounts, depending upon the bit rate of the song.

To be more visually pleasing, the decoder message will be posted once when the menu receives a PH event and then twice more in 100 ms increments by the soft timer. These additional decoder messages will update the LCD current seconds display field 10 times a second, instead of 3.33 times a second.

A counter will count the number of PH_XX events and will change the SecondstoSkip variable according to the table above. The values in this table are the default values, they are defined in the project.inc file as:

	1st phase (0-15 PH_XXs)	2nd phase (15-30 PH_XXs)	3rd phase (> 30 PH_XXs)
t_{sec} = Seconds to skip	1 second	3 seconds	6 seconds

Table 10.

```
SECONDS_TO_SKIP      equ   1
SECONDS_TO_SKIP1     equ   3
SECONDS_TO_SKIP2     equ   6
```

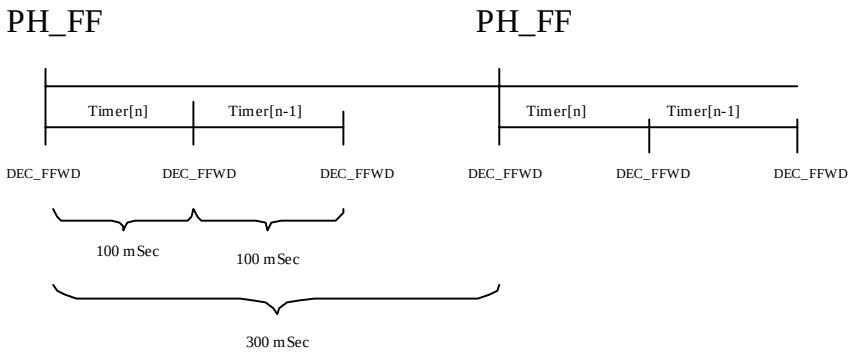


Figure 17. FFWD/RWND Event Timing diagram

The hold time will stay constant at 300 msec. Two new menu message will be created that post a decoder FF/RWD message. One soft timers will be set when a PH_FF/RWD comes in for 100 mSec and then once again after another 100 mSec has passed, as shown in the figure above, which will post the new message created MENU_MSG_SEND_RWND or MENU_MSG_SEND_FF respectively.

A counter will be added to keep track of the number of PH_XX events that have occurred, when the counter goes above the value of 15 then the global variable SecondstoSkip will be changed from SECONDS_TO_SKIP second to SECONDS_TO_SKIP1. Then the same would be done for 30. These values are stored in the project.inc file for the user to easily modify.

```
FIRST_TIME_BOUNDARY      equ  15 ;
SECOND_TIME_BOUNDARY  equ  30 ;
```

11.1.6. A/B Mode

AB mode is only available in the music playset (not voice).

AB mode allows the user to mark two places in the track and loop playback between these two marks.

Current SDKs limit the AB loop to a maximum of 1 second to prevent decoder message overflows.

11.1.6.1. AB Decoder Messages

Initialize the decoder module.

```
CheckDecoderInit();
```

A single decoder message will toggle through the AB states as shown :

```
SysPostMessage(2,DECODER_AB_MODE);
```

COMPANY CONFIDENTIAL 9/26/03

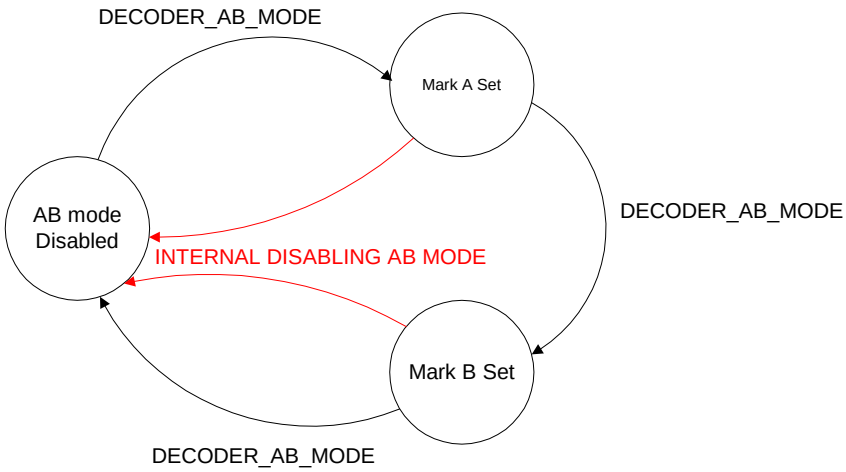


Figure 18. AB State Machine

Only the `DECODER_AB_MODE` message can set song marks enabling AB mode; however, AB mode will be disabled when the decoder module processes messages which cause the decoder to stop. Hence, a pause would not turn the AB Mode off nor would PH_FF/RW unless the beginning or end of a song was reached.

11.1.6.2. AB Status Flags

Routine provided in SDK example playerstatemachine.c to return AB mode status.

```
reentrant int PSM_GetABMode(void);
```

Outputs status bits in `Fg_wDecoderCSR`.

DECODER_B_SET : Both marks are set and decoder is looping. Display shows A-B icon. Uses the following system variables to determine state:
`g_wDecoderCSR & DECODER_B_SET`

DECODER_A_SET : Mark A is set; waiting for mark B before looping. Display shows A- icon
`g_wDecoderCSR & DECODER_A_SET`

0 : AB mode is disabled
`!(g_wDecoderCSR & (DECODER_A_SET|DECODER_B_SET))`

11.1.7. Volume Control

11.1.7.1. SDK2.1xx Multi-Stage Volume Control

Use these functions to interface with the multi-stage volume control. See Section 15.1. Multi-stage Volume Control for more details on operation and configurable settings.

SysIncrementVolume()

SysDecrementVolume()

11.1.7.2. SDK 2.0xx Single Stage Mixer Volume Control

Use the mixer module to adjust the master volume control register. SDK examples adjust by one bit (1.5dB) for each button press.

Decrement the master volume control by 1.5dB.

COMPANY CONFIDENTIAL 9/26/03


```
SysPostMessage(2,MIXER_MASTER_INCR);
```

Increment the master volume control by 1.5dB.

```
SysPostMessage(2,MIXER_MASTER_DECR);
```

11.1.8. PlayerLib Extracted Track Information

SDK examples use the variables in this section to display the current track information after a FF, RW, STOP or MENU_DECODER_STATE_CHNG event.

The track information is gathered by calling any one of the three PlayerLib functions: PlayerLib_GetCurrentSong(), PlayerLib_SkipNextSong() and PlayerLib_SkipPreviousSong().

In current SDKs, the track information in this section is called when processing parser messages. However the menu level does not typically access the parser directly but rather use decoder module messages which in turn post parser messages.

11.1.8.1. Track Name

All track names are accessed by the 8.3 short filename FAT entry. The long filename entry is not extracted in current SDKs. If the developer wants to display the long filename a routine to extract the long filename (usually in UNICODE) must be written with appropriate support for displaying UNICODE.

**m_pFilename*: This is a pointer to the packed path and file name. This path and short file name is sent to the decoder as the file to be played. If an ID3 tag does not exist, the playlist will strip the path and use the short file name as the Song Title leaving Artist and Album Title blank. During playback, this variable is used no matter on the format of the supported playback file.

11.1.8.2. Track Metadata Information

Metadata refers to the file information embedded in the track. The storage space for the track metadata is limited to 30 characters for all decoders.

Both ID3 version 1 and version 2 tags are support, only Artist, Song Title, Album Title. Extraction is performed by the PlayerLib function called PlayerLib_GetMetaData(). This function then determines the extension of the filename and calls the appropriate metadata function: GetMp3MetaData(), GetWmaMetaData(), or GetWavMetaData(). Each of these functions performs the necessary file lookups to fill out the metadata structure with the Artist, Title, Album, sample rate, song length, etc.

Fg_wSongTitle : MP3 - ID3v1/v2 Title; WMA - content property Title ; ADPCM - filename. If Title metadata is blank the short filename is copied here.

Fg_wSongArtist : MP3 - ID3v1/v2 Artist; WMA - content property Artist ; ADPCM - not used.

11.1.8.3. Track Number

Only the supported extensions are counted as tracks in the playlist.

Directory refers to root directory (\) for music mode and voice directory (\voice) for voice mode.

Fg_wCurrentSongNumber : Current track number in playlist.

Fg_wTotalSongCount : Total number of songs on all devices in the system.

Fg_wDirectorySongNumber : Current track number in directory.

COMPANY CONFIDENTIAL 9/26/03

11.1.9. Decoder Extracted Track Information

11.1.9.1. Encoder Track Time

The encoder will send a MENU_SONG_TIME_CHNG message to the menu after these variables are updated every second.

Current position of track being recorded.

Fg_iEncCurrentSeconds

Total time available for recording based on remaining memory:

Fg_iEncTotalSeconds

The hours and minutes are calculated from these variables and then displayed.

11.1.9.2. Decoder Track Time

The decoder will send a MENU_SONG_TIME_CHNG message to the menu after these variables are updated every second.

The VBR time functionality based on Xing headers has been implemented in SDK2.320. If the VBR track has a Xing header (most common) then, accurate time information will be displayed even after FF/RW.

Current position of track being played:

Fg_wSongCurrentMinutes, Fg_wSongCurrentSeconds

Total track time of current track:

Fg_wSongTotalMinutes, *Fg_wSongTotalSeconds*

MP3 tracks do not display the total time until the track is played.

11.1.9.3. Track Encoding Properties

The following bit and sample rate variables exist in application space and are only valid when the corresponding decoder is running.

Fbitrate_o **:** MP3 bitrate of current track, stored in X memory.

Fg_wSongInvBitRatePtr : The inverse bitrate of MP3 and ADPCM tracks is stored in a Y system (always resident) variable.

Fsampfreq_o : MP3 sample rate of current track is stored in X memory.

Ffilehdr_bitrate **:** WMA bitrate stored in Y memory.

Ffilehdr_sample_rate **:** WMA sample rate stored in Y memory.

11.2. Playset

function: _reentrant void ChangePlaySet(INT iPlaySet)

filename: \stmp3xxx_sdk\projects\sdk\lcdexample\player\system\ChangePlaySet.c

The playset is defined as either PLAYSET_MUSIC or PLAYSET_VOICE. The function ChangePlaySet() will set the playlist to either music mode of voice mode. It will also stop the decoder and point to the current song in the playset. The function Playlist_GetPlaySet() can be used to check for which playlist the player is set up.

When the playset is in voice mode, the playlist will be looking for voice files inside of the voice directory, it will not see any music files (*.mp3, *.wma) and therefore not list them on

the playlist. However when the playset is in music mode, the playlist will accept both voice (*.wav) and music (*.mp3, *.wma) files into the playlist from the root directory.

The directories default to root for the music playset and \voice for the voice playset, however they can be changed.

11.3. Play Mode

The playmode is controlled by the playlist command Playlist_ShuffleList().

In the SDK examples, the playmode menu variable is a saved setting (Fg_iPlayModeSetting). The Playlist_ShuffleList() command must still be posted prior to playback in order to set the playmode. The function SendPlayMode() is called during player initialization in mainmenu.c. in order to call the playlist command. This function is in a different codebank so must conform to the prototype required by SysCallFunction(). Refer to Section 6.3.3. SysCallFunction

Default operation without configuring a playmode is to play all the tracks in order of the playlist once. Start with track 1 and stop at track N, where N is the total number of valid music tracks. Invalid tracks will be skipped and the next valid track will continue playing until the end of the playlist. The two parameters which decide the playmode the player is in are g_iPlaylistRepeat and the first parameter of Playlist_ShuffleList(). The PlayList supports the playmodes as follows, also listed are the two parameters' values.

Music Playmode	g_iPlaylistRepeat	Shuffle parameter
Repeat Off	PLAYLIST_REPEAT_OFF	FALSE
Repeat Song	PLAYLIST_REPEAT_ONE	FALSE
Repeat All	PLAYLIST_REPEAT_ALL	FALSE
RandomOn	PLAYLIST_REPEAT_OFF	TRUE
Random Off	PLAYLIST_REPEAT_OFF	FALSE
Repeat All Random	PLAYLIST_REPEAT_ALL	TRUE

Table 11. MUSIC PLAYMODES

Note: REPEAT SONG and RANDOM are not supported -- REPEAT SONG will execute first without consideration for RANDOM. Menus should not allow this combination.

The functionality of the playmode applies for both the voice and music mode, it does not distinguish between the two.

11.4. Voice, FM, and Line-In Record

11.4.1. Record Settings

The record settings have been expanded for SDK2.320. The recordsettingsmenu.c is the menu to control the UI needed to select the settings specific to each source. Within the recordsettingsmenu.h, the record settings structures are defined as shown below:

- RecorderSettings - Structure containing record property members.
  - m_EncoderNo - Stores encoder type.
  - m_iChannels - Record in Mono or Stereo.
  - m_iDestinationDevice - Record to internal or external media

  - m_iSampleRateInHz - Record sample rate.
- g_RecorderSettings[] - Array used to store values of the RecorderSettings structure for each of the sources (Mic, FM, Line-in).
- g_ADCsource - Stores current source and used with g_RecorderSettings[] to select or display the appropriate settings.

Also defined in the recordsettingsmenu.h, are the available settings for each property.

11.4.2. Stmp Hardware Setup

The ADC and mixer on STMP3xxx must be setup in order to record from the correct source. Since all three sources are enabled in SDK2.320, a build option was created for the 100-pin (2 sources) and the 144-pin (3 sources). The 100-pin package is used as a default and will only enable 2 sources, 1 for Mic and 1 for Line-In1 (called SOURCE_FM in code).

Note: See the 144-pin comment within the player.mk for information on how to enable the 3 sources and 144-pin specific build options.

The following text outlines the required steps to set-up the ADC and mixer for recording.

Setup volume level for the selected source instead of using DAC volume level. When recording from FM, this is not necessary since the current output volume will be used.

```
SysPostMessage(3,MIXER_MIC_SETLVL,Gain);
```

where, Gain is a value from +12dB to -34.5dB -- constants are defined in regs34x0.h for Mic or Line-In (i.e. HW_MIXMICINVR_GN_ZERO_SETMASK)

UnMute the volume level, which may not be needed if the source was already used (i.e. FM).

```
SysPostMessage(2,MIXER_MIC_UNMUTE);
```

Add microphone boost (fixed +20dB) if recording from Mic-In

```
SysPostMessage(2,MIXER_MIC_BOOST);
```

Set ADC recording input

```
SysPostMessage(4,MIXER_REC_SELECT,LChSource,RChSource);
```

where, LChSource and RChSource are Mic-In, Line-In1 or Line-In2 bit definitions in datasheet -- constants are defined in regs34x0.h (i.e. HW_MIXRECSELR_SL_MIC_SETMASK)

Set gain stage to adjust the analog signal before reaching the ADC

```
SysPostMessage(3,MIXER_ADC_SETLVL,AdcGain);
```

where, AdcGain is the combined gain setting for both channels. Each channel can have a gain setting between +22.5db to 0dB -- constants are defined in regs34x0.h

```
(HW_MIXADCGAINR_GL_22P5_SETMASK |
HW_MIXADCGAINR_GR_22P5_SETMASK)
```

The ADC gain is lowered for FM and Line-In Recording to prevent clipping.

Unmute the ADC gain stage

```
SysPostMessage(2,MIXER_ADC_UNMUTE);
```

Note: There are not SDK mixer module messages to abstract the remaining actions.

Power up the ADC (DAC could be powered down also) in the Mixer Power Down Register

```
HW_MIXPWRDNR.B.PR0 = 0;
```

COMPANY CONFIDENTIAL 9/26/03

3410 ONLY -- Enable ADC clocks

```
HW_ADCCSR.B.CLKGT = 0;
```

11.4.3. Encoder Module Setup

To transfer the record settings to the encoder, the EncSetProperties() function is used. Each of the properties Struct members are filled with the value based on the stored properties in the g_RecorderSettings[] array. Once the properties are filled, the EncSetProperties() function is called to pass the values to the encoder.

RETCODE EncSetProperties(propertiesStruct)

// See \inc\encoderproperties.h for structure definition

The function will return an error code (non-zero) if the specified rate is invalid or if the encoder is already running.

Several macros, made to look like functions, exist for retrieving the current encoder properties:

EncGetSampleRateInHz()

EncGetDevice()

EncGetFilename()

EncGetEncodingMethod()

EncSetProperties is a funclet which doesn't have to be included at built time. If the funclet is never called then the default properties will be used (8kHz Sampling).

The encoder type must be setup so that the proper encoder will be loaded by the executive. Anytime the module type is changed the EVENT_INIT flag must be set in the module table Event Signal so the initialization routine for that module type will be run.

This function call will set the initialization flag in Event Signal for the encoder module.

```
EncoderForceInit();
```

This function call will load the IMA ADPCM encoder resource number in the encoder module table entry. There is currently only a single encoder, but the architecture is designed for expansion and this setup will allow additional modules to be added without changing the system.

```
SysSetEncoder(ENCODER_TYPE_ADPCM_IMA);
```

This message loads the application space with encoder resources and initiates recording. Due to this limitation no other encoder message should be posted prior to ENCODER_RECORD.

```
SysPostMessage(2,ENCODER_RECORD);
```

11.5. GEQ

The MP3 library without EQ (coremp3_NoVolTone.a) is linked in the build.

The Graphical Equalizer (GEQ) is part of the foreground processing chain that is controlled by a messaging module API. The current implementation of the GEQ uses 5 programable bandpass filters.

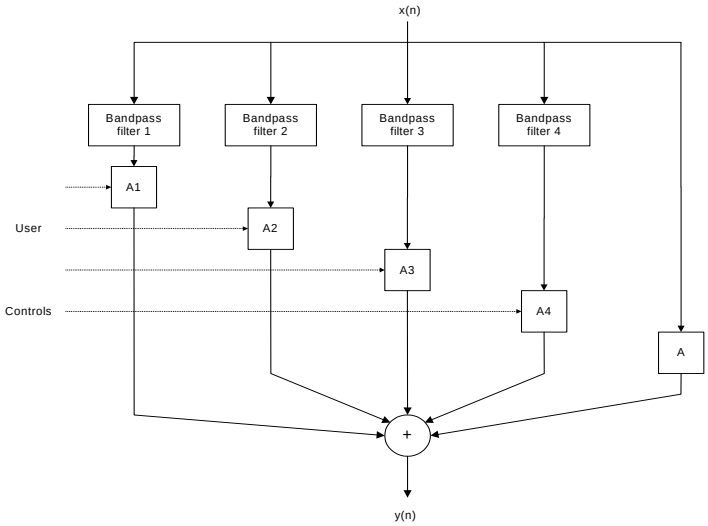


Figure 19. GEQ Control

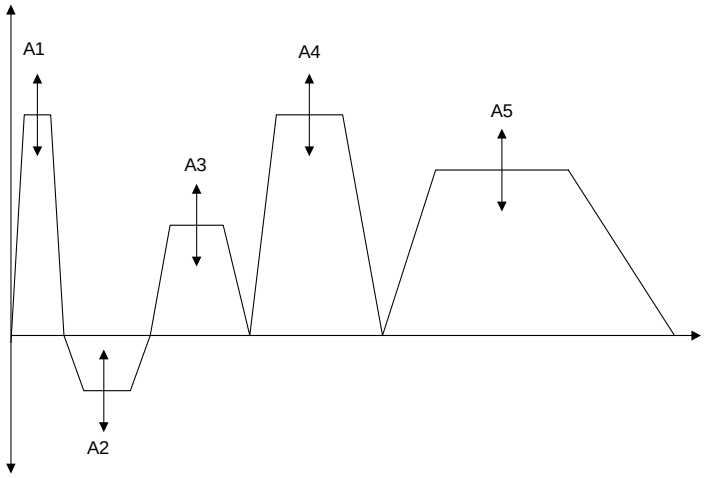


Figure 20. Example GEQ filters

COMPANY CONFIDENTIAL 9/26/03

11.5.1. GEQ Message Definitions

# Msg Words	Message Definition	Arguments	Description
3	GEQ_SET_EQ	TRUE/FALSE	Enables or Disables the Equalizer
2	GEQ_SET_EQ_LEGACY	None	Disables the GEQ
12	GEQ_SET_PARAM	(1/2Q) for 5 filters center freq for 5 filters	Q range: 1/(2Q) < 1 Center frequency range: 20 to 20000 Hz ** Combinations resulting in non-feasible filters will be accepted, but coefficients will be set to zero.
4	GEQ_SET_GAIN	Band Number Band Gain	Sets the band gain for the filter number
7	GEQ_SET_ALL_BAND_GAINS	Gain for each 5 bands	Sets gain for all the filters. The order is consistent with the order of the setup parameters (GEQ_SET_PARAM)
3	GEQ_SET_COEFFICIENTS	Sampling rate	Used to calculate the coefficients of the filter based on current track sampling rate. Used by decoder modules only.
2	GEQ_GET_SETTINGS	None	Call this function to get the center and level for all bands in the EQ. This could be used at shutdown to acquire user defined EQ settings to be saved to flash. Current SDK examples provide an alternative method for saving EQ presets.

Table 12. GEQ Messages

Filter gain parameters can be set between -14dB to +14dB in 0.5 increments. BUT the values used in the messaging API is multiplied by 2. So, the numeric range for the messages is -28 to +28 with increments of 1. (i.e. -28 = -14dB and -27 = -13.5dB)

Note: Band gain is limited by Fg_VolumeBias -- see Section 15.1. Multi-stage Volume Control

11.5.2. GEQ Initialization

In order to properly initialize the GEQ module, 2 messages must be sent before audio processing.

```
GEQ_SET_PARAM & GEQ_SET_ALL_BAND_GAINS
```

GEQ_SET_PARAM: is required to setup the filters

GEQ_SET_ALL_BAND_GAINS: sets up the attenuation and initializes g_VolumeBias.

11.5.3. SDK GEQ Example Usage

mainmenu.c

Send message to setup bandpass filters with 1/(2Q) = 0.99 or Q = 0.505 and center frequencies at 62,250,1k,4k,16k.

```
SysPostMes-
sage(12,GEQ_SET_PARAM,0.99,0.99,0.99,0.99,0.99,62,250,1000,4000,16000);
```

Send message to initialize gain. In this example the previous or default EQ is restored via SendEQ() which sends the GEQ_SET_ALL_BAND_GAINS message.

```
SysCallFunction(RSRC_EQ_MENU_CODE_BANK,SendEQ,
g_iEqSetting,0,0);
```

eqmenu.c

COMPANY CONFIDENTIAL 9/26/03

Only adjusts cut and boost of the filters, where dB settings are:

`iBandGain1` (filter1 corresponds to 62Hz center frequency) = +9db

`iBandGain2` (filter2 corresponds to 250Hz center frequency) = -2dB

`iBandGain3` (filter3 corresponds to 1kHz center frequency) = -1dB

`iBandGain4` (filter4 corresponds to 4kHz center frequency) = +7dB

`iBandGain5` (filter5 corresponds to 16kHz center frequency) = +7dB

```
case EQ_ROCK     ://ROCK
        iBandGain1=18;
        iBandGain2=-4;
        iBandGain3=-2;
        iBandGain4=14;
        iBandGain5=14;
        break;
SysPostMes-
sage(7,GEQ_SET_ALL_BAND_GAINS,iBandGain1,iBandGain2,iBandGain3,iBandGain
4,iBandGain5);
```

11.6. Delete Files

FATWrite overlay must be loaded to delete files. In the case of MMC be sure the device driver includes the write function (include mmcddwrite.mk instead of mmcdd.mk)

Stop the decoder/encoder before browsing or deleting files.

Point the parser to the correct playset in order to access the files in each playset; either PLAYSET_MUSIC or PLAYSET_VOICE

Use DECODER_NEXT_SONG message to get the next track name.

FSFileDelete() is called to delete the selected file using the filename.

Unload FATWrite module SysUnLoadFatWriteCode() to allow other overlay apps to be loaded.

Force decoder to reload app space: DecoderForceInit();

Update valid song count: parser message PARSER_DEVICE_ENUMERATE

Force decoder to reload app space: DecoderForceInit();

11.7. FM Tuner

The FM tuner firmware provides a manual station searching, auto station searching, scanning for presets, and tuning to a preset. The output of the FM tuner is analog signal and is connected to either FMIN or LINEIN1 depending the package type of the STMP chip. For the 100-pin TQFP package, the FM Tuner is connected via LINEIN1.

There are 3 levels of the software for this. The fmtunermenu.c covers the user interface with the displays.c. The buttons are translated into tuner control messages sent to tunermodule.c. The list of messages are shown in table 1. The tunermodule.c calls the tuner drivers for either the TEA5767 or TEA5757 to do all the tunings. There is a single message from the tuner drivers to the menu.

If the all the presets are empty, the FM tuner performs the preset scanning to store up to 10 preset stations. The TEA5757 stores 10 strongest stations while the TEA5757 stores the first 10 stations found.

The audio is muted during the tuning and continue muted for additional 300 mSec.

The audio is muted for 3 seconds during the tuner is powering up (entering fmtunermenu).

The TUNER_GET_STATES is sent every 2 seconds to update the stereo pilot unless it is commented out.

11.7.1. FM Tuner Module Messages

Messages to tunermodule.c	Parameters	Description
TUNER_TUNE_FREQUENCY	Frequency in KHz	Set PLL to the given frequency
TUNER_TUNE_MANUAL_UP		Tune up one grid **(1)**
TUNER_TUNE_MANUAL_DOWN		Tune down one grid **(1)**
TUNER_TUNE_SEARCH_UP		Search up for the next FM station **(2)**
TUNER_TUNE_SEARCH_DOWN		Search down for the previous FM station **(2)**
TUNER_FORCED_MONO	TRUE or FALSE	Turn off the MPX decoder
TUNER_POWER_ON		Turn on the FM Tuner
TUNER_POWER_OFF		Turn off or put the FM Tuner in stand-by mode
TUNER_SET_SENSITIVITY	0 to 100	Set the searching sensitivity
TUNER_GET_STATES		Get the tuner's stereo pilot state and current frequency.
TUNER_PRESET_STATION		Clear all the presets and begins scanning for FM stations
TUNER_TUNE_TO_PRESET	Preset number	Tune to the given preset (3)
TUNER_SET_PRESET	Preset number	Store the current tuned frequency into the given preset number
TUNER_ERASE_PRESET	Preset number	Clear the current preset
TUNER_RESET		na
Message to menus		
MENU_TUNER_TUNED		Signal the menu that the tuner has finished a task

Table 13. FM Tuner Module Messages

NOTES:

(1) It rolls over at the end of the FM band.

(2) If g_bSearchEndAtBandLimit is false, it rolls over at the end of the band

(3) If the preset is empty, the frequency shows 87.5MHz and the audio is muted

11.7.2. FMTuner Global variables:

WORD g_wCurrentFrequency: tuner's frequency in KHz.

BOOL g_bTunedStereoStation: stereo pilot status

BOOL g_bSearchEndAtBandLimit : controls the searching for FM station

WORD g_wCurrentPreset : current preset. (Equals 0 when tuned out of the preset).

FMTunerPreset g_astPresetStations[NUMBER_OF_PRESETS]: stores the frequency and signal strength of the presets.

11.7.3. Example FM Tuner UI

The following table describes the user interface implemented in the SDK LcdExample.

Button name	Press and release	Press and hold
Next	Manually tune up one grid	Search up to the next FM station **(1)**
Previous	Manually tune down one grid	Search down to the previous FM station **(1)**
Play	Mute/unmute the audio	Mute/unmute audio on release of the button
Stop	Not applicable	Counting down toward shut down. Audio is muted while the button is pressed.
Menu	Exit to the main menu	Exit to the main menu on release of the button.
Mode	Select the next preset FM station	Same as the p/r on the release of the button
A-B	Select the previous preset FM station	Same as the p/r on the release of the button
EQ.	Not applicable	Not applicable
Record	Store the tuned frequency in the last selected preset	Same as the p/r on the release of the button
Erase	Erase the last selected preset	Same as the p/r on the release of the button
Vol +	Increase the audio volume	Repeatedly increase the audio volume
Vol -	Decrease the audio volume	Repeatedly decrease the audio volume

Table 14. LcdExample FMtuner User Interface

Note: (1) the searching command is sent every 5 seconds while the button is held down.

11.7.4. Customizing the FM Tuner

- Edit player makefile to include either Philips5757.mk or Philips5767.mk
- Edit the NUMBER_OF_PRESET in resource.inc to change the number of pre-sets
- TEA5767 selectable functions by defining values in resource.inc or project.inc
- add '#define FM_TUNER_HCC 1' to enable high cut control
- add '#define FM_TUNER_SNC 1' to turn on the stereo noise canceling
- add '#define FM_TUNER_SOFT_MUTE 1' to set soft mute on
- add '#define JAPAN_FM_BAND 1' to select Japan FM Band

- add '#define FM_TUNER_US_DEEMPHASIS 1' to select 75 uSec.
- TEA5757 has no customizable feature
- Add 'FMTUNER_ON_LINE2_IN equ 1' to project.inc to select the FMIN on the 144-pin TQFP.

11.7.5. Improving FM Tuner Performance

The NAND accesses to load overlays have proven to interfere with FM tuner reception. To help reduce the noise introduced by the NAND access, eliminate as many overlays while in FMtuner mode. The 2 second sofftimer battery check and display updates should be reworked to minimize overlays. A decoder process may also be running in the background unnecessarily.

Also reduce the system speed during FMtuner mode to IDLE.

SigmaTel is working on a final solution with optimum FM tuner performance that will be provided in the SDK when complete.

12. PLAYLIST API

12.1. PlayList Data Structure

```
Struct SongFileInfo
{
WORD     m_wTrack;
WORD     m_wDeviceID;
WORD     m_wBufferLength;
_packed char *m_pFilename;
};
```

*m_pFilename: The filename is chosen to be a packed character due to space requirements (could be quite long with directory names, etc)

```
PLAYLIST_SUCCESS                    0
PLAYLIST_FILE_ERROR                 1
PLAYLIST_END_OF_LIST                2
PLAYLIST_TRACK_NOT_FOUND            3
```

12.2. Required PlayList Library Entry Points

The core system player state handler requires three entry points of the playlist library to link and function properly. These functions are Playlist_GetCurrentSongFile(), Playlist_GetNextSongFile(), and Playlist_GetPreviousSongFile(). These songs are used to iterate the playlist and move from one song to another.

12.2.1. Playlist_GetCurrentSongFile()

RETCODE _reentrant Playlist_GetCurrentSongFile(Struct SongFileInfo*)

Input: Struct SongFileInto * with m_iBufferLength and m_pFilename initialized.

Output:Struct SongFileInfo fully populated if return value is SUCCESS, else error code.

This function is called by the player state handler when going from stop state to start state. This function should either return PLAYLIST_SUCCESS, or PLAYLIST_END_OF_LIST. Any other error conditions should be handled within the implementation. The player state handler will stop the decoder on any return code other than PLAYLIST_SUCCESS.

12.2.2. Playlist_GetNextSongFile()

RETCODE _reentrant Playlist_GetNextSongFile(Struct SongFileInfo *);

Input: Struct SongFileInfo * with m_iBufferLength and m_pFilename initialized.

Output:Struct SongFileInfo fully populated if return value is SUCCESS, else error code.

This function is called by the core system when one song is ending, and another song needs to be opened by the decoder. This function should either return PLAYLIST_SUCCESS, or PLAYLIST_END_OF_LIST. Any other error conditions should be handled within the implementation. The player state handler will stop the decoder on any return code other than PLAYLIST_SUCCESS.

12.2.3. Playlist_GetPreviousSongFile()

RETCODE _reentrant Playlist_GetPreviousSongFile(Struct SongFileInfo*)

Input: Struct SongFileInto * with m_iBufferLength and m_pFilename initialized.

Output: Struct SongFileInfo fully populated if return value is SUCCESS, else error code.

This function is called by the core system when one song is ending, and another song needs to be opened by the decoder. This function should either return PLAYLIST_SUCCESS, or PLAYLIST_END_OF_LIST. Any other error conditions should be handled within the play list implementation. The player state handler will stop the decoder on any return code other than PLAYLIST_SUCCESS.

12.3. Optional PlayList Library Entry Points

The following entry points are optional and are given as an example as ways the user interface could interact to control the play list.

12.3.1. Playlist_Init()

RETCODE _reentrant Playlist_Init(void)

Inputs: Mode in which the playlist should operate. The meaning of the values of this parameter is entirely up to the playlist developer.

Output: SUCCESS if the playlist was initialized properly.

12.3.2. Playlist_GetSongFileByNumber()

RETCODE _reentrant Playlist_GetSongFileByNumber (Struct SongFileInfo*)

Inputs: Struct SongFileInto * with m_iBufferLength, m_pFilename, and m_iTrack initialized.

Output: Struct SongFileInfo fully populated if return value is SUCCESS, else error code.

The user interface could use this mechanism to browse the play list, since it would not upset the internal counters of the play list library. The user interface could also browse the play list by indexing into an array, or whatever mechanism the developer felt was appropriate.

12.3.3. Playlist_SetSongByNumber()

RETCODE _reentrant Playlist_SetSongByNumber(WORD wTrack)

Inputs: Track number to set as the current song track number.

Output:SUCCESS if the playlist can be set to the input track number.

12.3.4. Playlist_SetMode()

RETCODE _reentrant Playlist_SetMode(WORD wMode)

Inputs: New mode in which the play list should operate.

The meaning of the values passed in is entirely up to the play list developer. Some examples would be MODE_REPEAT_ONE, MODE_REPEAT_ALL, MODE_SHUFFLE, MODE_NORMAL, etc. This would affect how the PlaylistGetnextSongFile() would choose its next song.

Player State Library (all functions have the SysCallFunction compatible prototypes and reside in RSRC_PLAYER_STATE_HANDLER code bank)

12.4. Player Library

12.4.1. Goto Track

SDK 2.320 supports resuming from the beginning of a saved track. The lcdexample has been modified when switching between music and voice menus, to save the last track

played in that menu and restore when the menu is returned. This functionality uses the playlists, so only music files can be restored (mp3, wma, pcm, imadpcm and msadpcm).

For example if a user is listening to track 3 in music mode, then switches to voice mode, and returns back to music mode track 3 will be restored as the current track.

Two structures are used. One for music mode and other for voice mode. These store track numbers for each mode.

```
struct Bookmark{
  INT        m_iTracknum;
  DWORD      m_dwTrackPosBytes;
}g_MarkerMusic, g_MarkerVoice;
```

m_iTracknum stores the song number of currently playing track

m_dwTrackPosBytes : not used -- reserved for future expansion

In the SDK example, the saved track (`m_iTracknum`)is updated under the following conditions:

- Changing songs: the saved track is updated to match the currently playing track while playing or FF/RR while traversing through the list.
- Changing modes: the saved track is updated to match the last played track when the user switches between playsets
- Deleting files: If any track is deleted in music or voice mode, the saved track for that mode is reset to start from beginning.
- Voice record: No effect on saved track
- External media changes: If external media is ejected and it contains a saved track, it will be reset to start from beginning.

The following APIs are provided for the save and restore track functionality:

`PlayerLib_SetBookmark(int iCurrentPlayset, int iTracknum,int* pPtr);`

Stores iTracknum in g_MarkerMusic or g_MarkerVoice depending upon iCurrentPlayset.

`PlayerLib_GotoTrack(int iTracknum, int ignored, int* ptr);`

Advances to iTracknum

`PlayerLib_ResetBookmark(int iCurrentPlayset, int ignored, int* pPtr);`

Clears the track for iCurrentPlayset

`PlayerLib_GetBookmarkSongInfo(int iCurrentPlayset, int ignored2, int* pPtr);`

Advances the playlist to the saved track for iCurrentPlayset. This function internally calls PlayerLib_GotoTrack.

12.4.2. PlayerLib_SetState()

RETCODE _reentrant PlayerLib_SetState (int iState,int bWait,int*);

Inputs: iState is the new state the decoder should be set to.

DECODER_STATE_STOP	Enters the stop state from either pause or play.
DECODER_STATE_PLAY	Enters the play state from either stop or pause.
DECODER_STATE_PAUSE	Enters the pause state from either stop or play.

DECODER_STATE_TOGGLE Toggles between pause and play (or goes from stop to play)

Outputs: None (return value is meaningless)

12.4.3. PlayerLib_FastForward()

RETCODE _reentrant PlayerLib_FastForward (int bPlayDuring,int,int*)

RETCODE _reentrant PlayerLib_Rewind(int bPlayDuring,int,int*)

12.4.4. PlayerLib_SetSongName()

RETCODE _reentrant PlayerLib_SetSongName(int bStartPlaying, int, (struct SongFileInfo*));

Inputs:

Outputs:

12.4.5. PlayerLib_SetSongName()

RETCODE _reentrant PlayerLib_SkipToNextSong(int,int,int*);

12.4.6. PlayerLib_SkipToPreviousSong()

RETCODE _reentrant PlayerLib_SkipToPreviousSong(int,int,int*);

12.4.7. PlayerLib_EnablePlaylist()

RETCODE _reentrant PlayerLib_EnablePlaylist(int bTrueFalse,int,int*);

Causes the playlist handling to be disabled. When the decoder stops, it will not skip to the next song.

12.5. PlayList Implementation

Firstly there are several data structures and arrays that are of key importance with respect to the play list implementation. The following are the main arrays and structures that will be needed. More will be introduced as this document goes on.

```
struct FileEntry{
        m_iDecoder;  //determines which decoder to play the file with
        m_iDevice;   //determines which media the file is on
        m_iMode;
        m_pszFilename[];   //stores the SFN of the file
        m_pNext;           //a pointer to the next file
        m_pContents;
        m_pContainer;//a pointer to the directory for this file/directory
        m_iFCBEntry; //File Control Block number for a file/directory
}
```

– This data structure is the foundation for the play list API. The structure contains all of the necessary information required to find a song and play it. It is prototyped in playlist1internal.h. The information within the structure consists of the following:

`With the addition of Long File Name support this data structure contains a new Element m_iFCBEntry`. The above structure `sets` the stage for the playlists, which are actually just a set of a few arrays.

Notice: these arrays consist either of instances of the `FileEntry` struct or pointers to instances of the `FileEntry` struct.

- FileEntry `g_FileEntryPerDevice`[MAX_LOGICAL_DEVICES]

  - This is a special array of `FileEntry` structures. The elements of this array contain important information about the location of the ROOTs of all devices on a player. Types of player may include:
  - Smart Media and NAND (MAX_LOGICAL_DEVICES = 2)
  - MMC/SD and NAND (MAX_LOGICAL_DEVICES = 2)
  - NAND only (MAX_LOGICAL_DEVICES = 1)
  - Additionally there is the option to have any of the builds above use a multi-nand configuration. With this type of player 2 or more NANDs combine to make one logical device. For a NAND only multinand build this array would only have one entry even though two NANDS are used. The software abstracts the fact that there are really two NANDs so that the player can see it as only one drive.

- FileEntry `g_FileEntryPool`[PLAYLIST_MAX_FILES]
  - The `g_FileEntryPool` contains information on all play list files on a player. In the sdk release this array is setup to allow for a maximum of 256 files/directories, but can be changed to accommodate particular needs. This particular array is highly memory intensive. Each entry in the array requires 10 WORDS of memory. Increasing PLAYLIST_MAX_FILES by N will use N*10 more WORDS of memory!

Notice: only files that are in a playlist are in the `g_FileEntryPool`.

- FileEntry *`g_PlayList`[PLAYLIST_MAX_FILES]
  - This array contains pointers to FileEntries in the `g_FileEntryPool`.
- FileEntry `g_iInternalList`[PLAYLIST_MAX_FILES]
  - This array contains indexes into `g_PlayList`. This is the array that is used to determine the actual order of playing files.

The following is a discussion of how these structures are populated when the player starts up. This is handled in the **`Playlist_BuildContentTree()`** function. Below is a flow chart with a synopsis of how the `g_FileEntryPool` [] is populated. This includes the calling of **`Playlist_PopulateDirectory()`** which is a reentrant, recursive function.

The `g_FileEntryPool` is populated in an order that may be somewhat counter intuitive yet is extremely practical once understood. It should be noted that songs as well as directories are stored in `g_FileEntryPool`. This is done so that recreating the directory path can be done in Q(n) time, where n is the depth at which the file exists. This will also save time and power, as less reads from the media will be needed when skipping from one song to the next. The directory structure illustrated below has been created on an example player. To better understand the order in which the `g_FileEntryPool` is constructed an explanation follows:

The orange zigzag line in the above picture illustrates the order in which the directories are populated into the `g_FileEntryPool`. This assumes that the folders were written to the device in the order of their naming (i.e. FOLDER1, FOLDER2…etc.).

While this does show the order in which they are populated into the `g_FileEntryPool` it fails to show that there will be entries between the FOLDER entries. For instance between the entries for FOLDER12 and FOLDER21 there will be three entries for files. In this case there will be entries for FILE2, FILE3, and FILE4. Within a folder the files are added to `g_FileEntryPool` in the order that they were copied to the Media from the

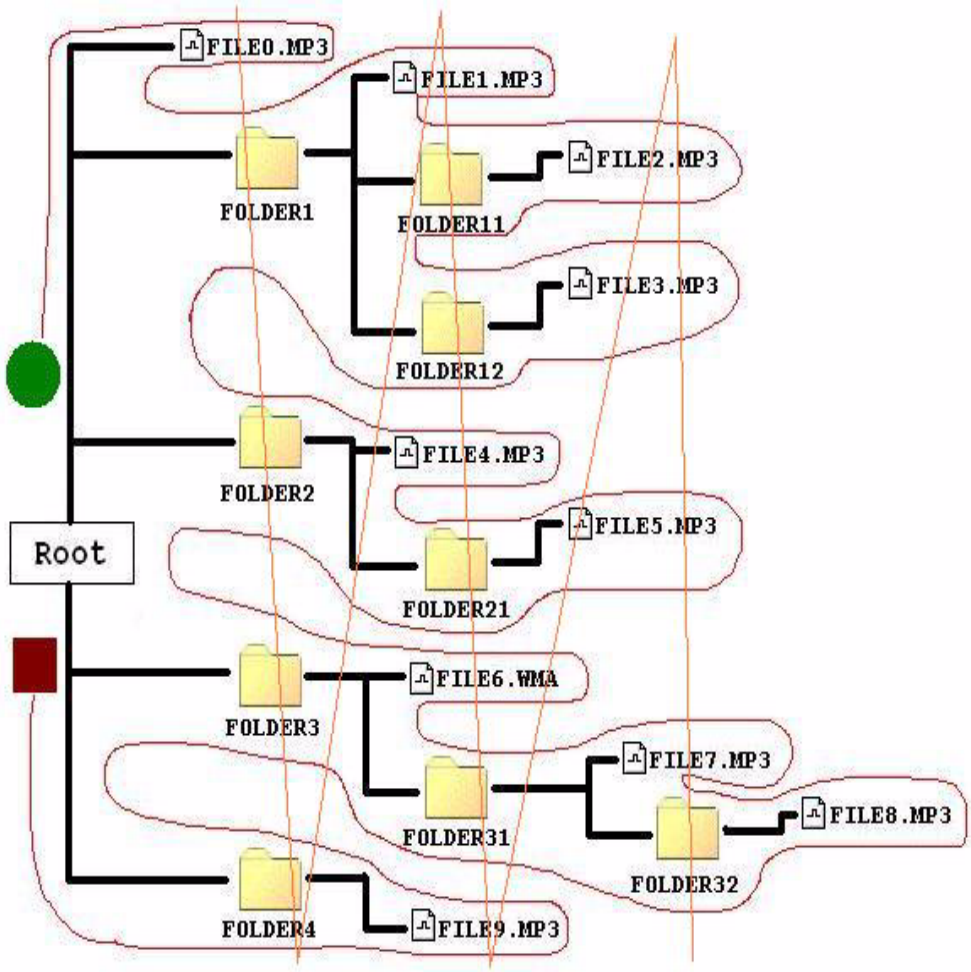


Figure 21.

host PC. This may not always be obvious as certain platforms take many liberties when copying files. The red traversal of the tree above (starting at the green circle and ending at the red square) indicates the order in which the files appear in `g_FileEntryPool`. This again will change if there are multiple files in one directory and the order in which they are copied changes. Luckily the manner in which `g_FileEntryPool` is not of great interest when adding a play list. It is included here for information only and is not necessary information for adding a play list. If you are feeling tenacious I would recommend trying a few directory structures out and paying attention to the order in which files/folders are copied and seeing if you can figure out how `g_FileEntryPool` will look in the debugger. After a few tries it should become apparent.

Below is a flow chart showing briefly how this population is accomplished. Below the flowchart are several screen shots taken from the BoxView debugger, which show the data on the STMP in the watch window. Each screen shot is followed with a brief description of the data.

When the player powers up a certain sequence of events happens to get the `g_FileEntryPool` populated. Firstly `Playlist_BuildContentTree()` is called from mainmenu.c calling `Playlist_Initialize`. This function in turn calls `Playlist_PopulateDirectory()` once for each logical device that is present in the player

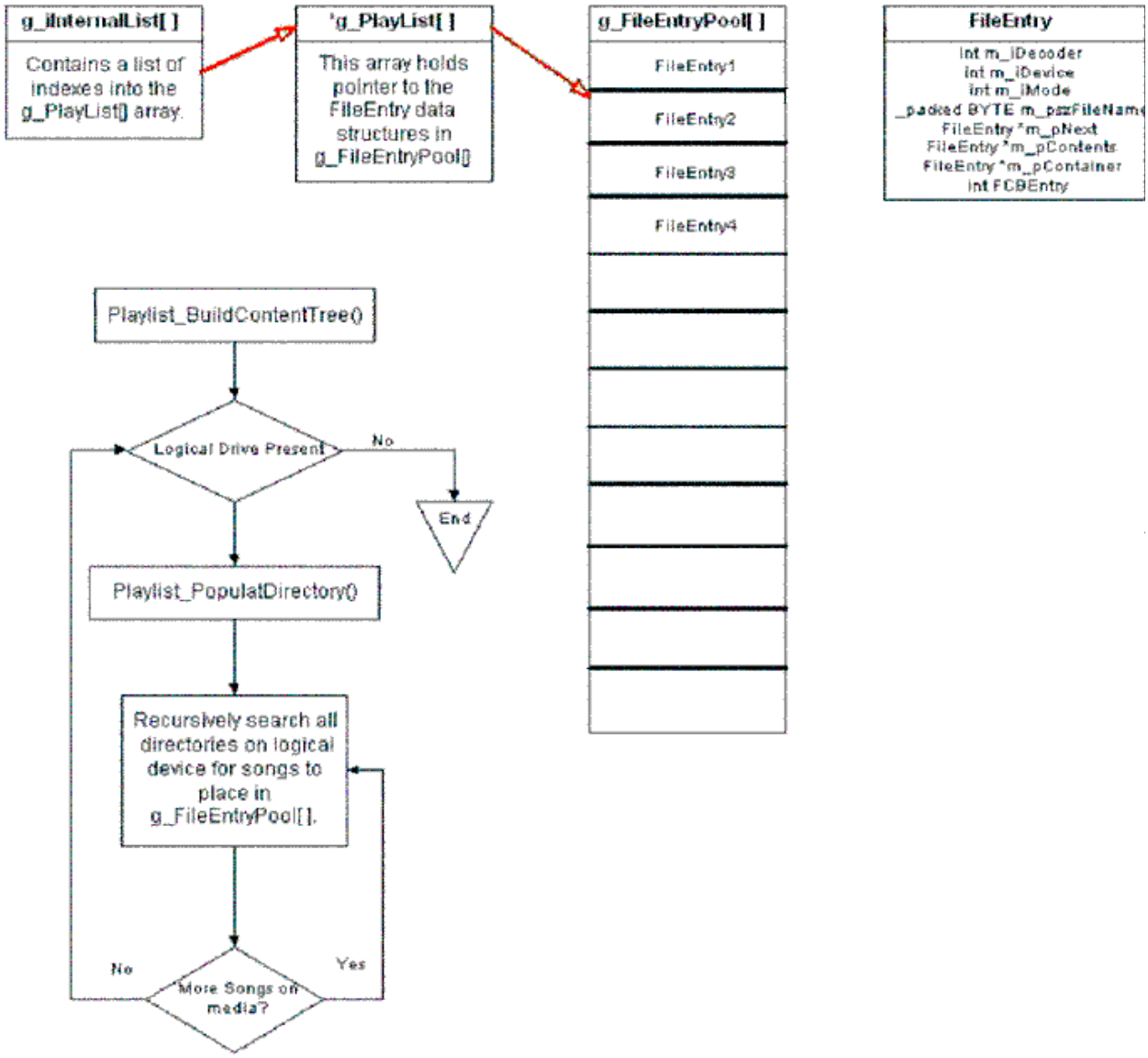


Figure 22.

build. For an MMC player it would be called twice. Playlist_PopulateDirectory() takes care of finding all of the files on the media. Once the `g_FileEntryPool` has been populated `g_PlayList` and `g_iInternalList` must be files based on which play list is currently selected. For the example player this defaults to **PLAYSET_MUSIC**. There are a few functions already defined that take care of this operation. The first function that is called is Playlist_FillQueue(). For each logical device in the player this function will call Playlist_AddToQueue(). Playlist_AddToQueue() is responsible for traversing `g_FileEntryPool` and adding entries into `g_PlayList` and `g_iInternalList` based on if the files should be included in the current PLAYSET (i.e. MUSIC or VOICE). More information on this process will be presented in the second part of this document. That part of this document explains in detail how to go about creating an entirely new play list on the player.

As an example of the above flowchart four files have been copied to the internal media (Device 0) of a player. Those files are (in order of being copied):

- 01-THE~1.MP3 01 - There's A Girl In Texas.mp3
- 06-BUB~1.MP3 06 - Bubble Toes.mp3
- 21-ALI~1.MP3 21-Alison Krauss - The Lucky One.mp3
- SIMON&~1.MP3 Simon & Garfunkle - Kodachrome.mp3

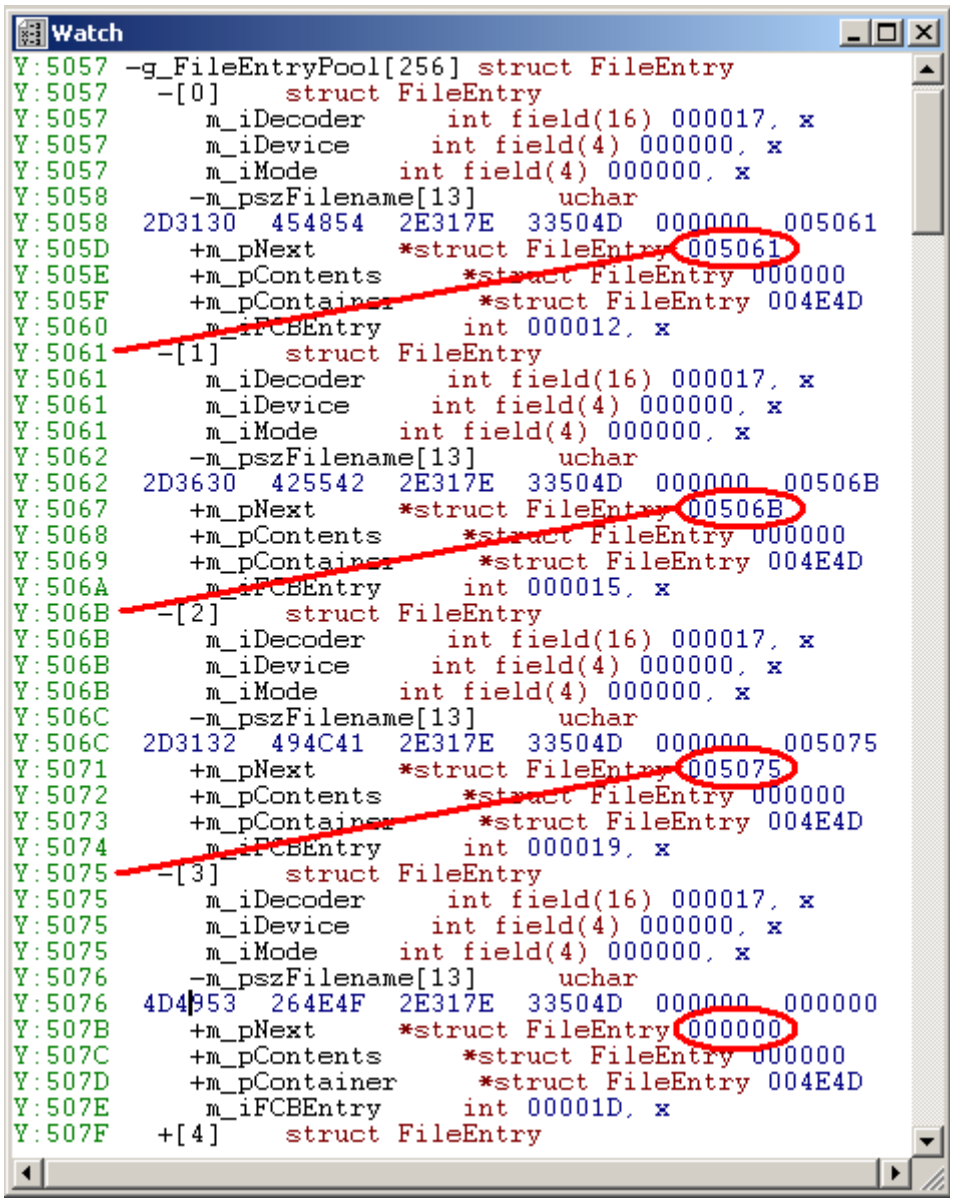

Watch
Y:5057 -g_FileEntryPool[256] struct FileEntry
Y:5057 -[0] struct FileEntry
Y:5057 m_iDecoder int field(16) 000017, x
Y:5057 m_iDevice int field(4) 000000, x
Y:5057 m_iMode int field(4) 000000, x
Y:5058 -m_pszFilename[13] uchar
Y:5058 2D3130 454854 2E317E 33504D 000000 005061
Y:505D +m_pNext *struct FileEntry 005061
Y:505E +m_pContents *struct FileEntry 000000
Y:505F +m_pContainer *struct FileEntry 004E4D
Y:5060 m_iFCBEntry int 000012, x
Y:5061 -[1] struct FileEntry
Y:5061 m_iDecoder int field(16) 000017, x
Y:5061 m_iDevice int field(4) 000000, x
Y:5061 m_iMode int field(4) 000000, x
Y:5062 -m_pszFilename[13] uchar
Y:5062 2D3630 425542 2E317E 33504D 000000 00506B
Y:5067 +m_pNext *struct FileEntry 00506B
Y:5068 +m_pContents *struct FileEntry 000000
Y:5069 +m_pContainer *struct FileEntry 004E4D
Y:506A m_iFCBEntry int 000015, x
Y:506B -[2] struct FileEntry
Y:506B m_iDecoder int field(16) 000017, x
Y:506B m_iDevice int field(4) 000000, x
Y:506B m_iMode int field(4) 000000, x
Y:506C -m_pszFilename[13] uchar
Y:506C 2D3132 494C41 2E317E 33504D 000000 005075
Y:5071 +m_pNext *struct FileEntry 005075
Y:5072 +m_pContents *struct FileEntry 000000
Y:5073 +m_pContainer *struct FileEntry 004E4D
Y:5074 m_iFCBEntry int 000019, x
Y:5075 -[3] struct FileEntry
Y:5075 m_iDecoder int field(16) 000017, x
Y:5075 m_iDevice int field(4) 000000, x
Y:5075 m_iMode int field(4) 000000, x
Y:5076 -m_pszFilename[13] uchar
Y:5076 4D4953 264E4F 2E317E 33504D 000000 000000
Y:507B +m_pNext *struct FileEntry 000000
Y:507C +m_pContents *struct FileEntry 000000
Y:507D +m_pContainer *struct FileEntry 004E4D
Y:507E m_iFCBEntry int 00001D, x
Y:507F +[4] struct FileEntry

Figure 23.

The circled items are the `m_pNext FileEntry` pointers. They allow for easy traversal of the `g_FileEntryPool` somewhat like a singly linked list.

Notice: The last file/directory in the `g_FileEntryPool` has an `m_pNext` pointer of value zero. This is used to indicate that it is the last file/directory.

The Short File Names (SFNs) can also be seen from the above screenshot. They are located underneath the `m_pszFileName` data members of the `FileEntry` structs. If you look closely you will notice that the file names are swizzled. Every 1st and 3rd characters are switched. This is how the File System expects them to be so it is done during population of `g_FileEntryPool` to save time later. Consult an ASCII chart if you are having trouble understanding the SFN encoding above. The next array that will be explored is the `g_PlayList` array.

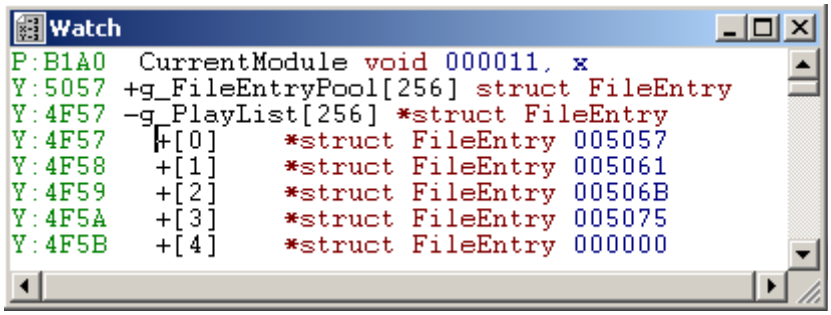

Watch
P:B1A0 CurrentModule void 000011, x
Y:5057 +g_FileEntryPool[256] struct FileEntry
Y:4F57 -g_PlayList[256] *struct FileEntry
Y:4F57 +[0] *struct FileEntry 005057
Y:4F58 +[1] *struct FileEntry 005061
Y:4F59 +[2] *struct FileEntry 00506B
Y:4F5A +[3] *struct FileEntry 005075
Y:4F5B +[4] *struct FileEntry 000000

Figure 24.

Looking to the above picture it is clear that the entries in the `g_PlayList` array are pointers to `FileEntry` entries in `g_FileEntryPool`. Take the time now to look at the above pictures and realize the relationship. Look at `g_PlayList[0]` and then follow the pointer to the `g_FileEntryPool`. Y:5057 is `&g_FileEntryPool[0]`. The `g_PlayList` array is populated through the play list API. It will contain different pointers depending on the play list that is loaded. When the voice file play list is loaded it will contain pointers to only .wav files that are in the root\voice\ directory. This is how the voice play list is defined by design. The last half of this document will describe the steps necessary to create an entirely new play list defined by your specifications. Similarly the music play list is defined to include all .mp3 and .wma songs in any folder on the player. You may be wondering how the player decides the order in which to play the files for a given play list. The final array to be discussed will explain that.

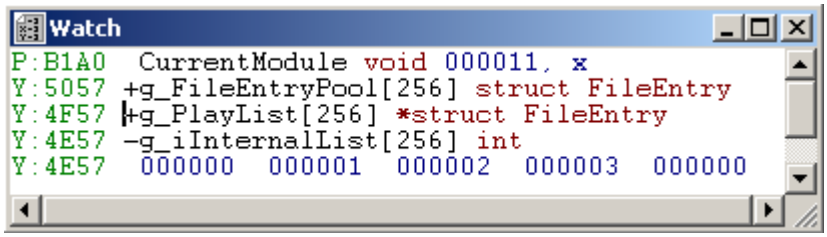


Figure 25.

The `g_iInternalList` array stores indexes into the `g_PlayList` array. This array is used to determine the order in which tracks are played. To shuffle the play list, for instance, is a simple random arrangement of the indexes in this array.

12.6. Playlist Customization

Now that you are confident with the way in which the play list works from a high level perspective it is time dig into the nuts and bolts of the API and create your own customized play list. For brevity the example that will follow will describe the process of creating a play list that includes .mp3 and .wma files that begin with the letter 'A'. While this type of play list may not be the most practical it will suffice to show all necessary steps to obtain a customized play list. The discussion that follows is a bit more involved then the above however it should be manageable.

There are several files that will be altered in our attempts to create a play list. A list of those files is below with a description of what needs to be added/changed.

- **system\common\playlist\playlist1.h**

This file contains a few `#define` statements as well as a few function prototypes. There are two `#define` statements used for play list functionality. They are in black text below. The additions necessary for the custom play list are in orange.

```
#define PLAYSET_FIRST     0
#define PLAYSET_VOICE     0
#define PLAYSET_MUSIC     1
#define PLAYSET_CUSTOM    2
#define PLAYSET_LAST      2
```

- **projects\sdk\lcdexample\player\menus\musicmenu.c**

This file contains all of the code for the music menu. This includes code to respond to button events and so forth. Below is a segment of code that works well with the two play sets that are defined in the SDK release but will not work with the custom play set. I will try and cover all of those places in this document but care should be taken to discover all of such places and make changed accordingly. The following code is in the **`MusicMenu()`** function.

```
if(Playlist_GetPlaySet() != PLAYSET_MUSIC)
{   //sdk2.1 (i.e. if currently not in music mode.)
    //initialize the music player state machine
    ChangePlaySet(PLAYSET_MUSIC);
}
```

To allow the above code to work with an additional play list we must change the conditional. Since our play list will only hold mp3 files we are certain that the music menu can play the songs in our play list. For this reason we only want to **`ChangePlaySet()`** to **PLAYSET_MUSIC** if the **PAYSET_VOICE** is currently selected. The code should be changed to look as follows:

```
If(Playlist_GetPlaySet() == PLAYSET_VOICE){
      ChangePlaySet(iEvent);}
```

This will allow for the music menu to use both the `PLAYSET_MUSIC` play list as well as the `PLAYSET_CUSTOM` play list. Next we must design a mechanism for switching between the play lists while in music mode. This can be done using either a button event or perhaps a menu item. For brevities sake we will use a button event to switch between the two. The button event to be used is `PR_RECORD`. For this reason it will not be possible to begin recording due to this button being pressed while in the music menu. This is due to the fact that the message will not be sent to the playerstatemachine for processing. The following code should be added in **`musicmenu()`**. Only orange code should be added.

```
case EVENT_BUTTON:
 //somebody pressed a button.
```

```
 //remember the last button event
   switch(g_wLastButton = gEventInfo.Button.wButtonEvent)
   {
      case PR_RECORD: //toggle between music and custom
          if(Playlist_GetPlaySet() == PLAYSET_MUSIC)
          {
             ChangePlaySet(PLAYSET_CUSTOM);
          }
          else{
             ChangePlaySet(PLAYSET_MUSIC);
          }
          bSendToPlayerStateMachine = FALSE;
          DisplayHint.I |=
(MUSIC_MENU_DISPLAY_ALL|(1<<DISPLAY_CLEAR_DISPLAY_BITPOS));
      break;
      … //There is more code already here.
}
```

- **system\common\playlist\playlist1.c**

The purpose of this file is to handle all play list related code for all play lists. We will be adding code to the function **`Playlist_AddToQueue()`**. That code is below in orange.

```
RETCODE _reentrant Playlist_AddToQueue(struct FileEntry
*pEntry)
{
    RETCODE rtn = !PLAYLIST_SUCCESS;
    if(g_iTotalTracks < PLAYLIST_MAX_FILES && pEntry-
>m_iDecoder != FILE_ENTRY_UNUSED)
   {//make sure we have room in the list and this file entry
is either a directory or a song
       if(pEntry->m_iDecoder==FILE_ENTRY_DIR)
      {//if its a directory, go through all of its contents
and add them to the list
           struct FileEntry *pCurrentEntry = pEntry-
>m_pContents;
           while(pCurrentEntry)
           {
               Playlist_AddToQueue(pCurrentEntry);
               pCurrentEntry = pCurrentEntry->m_pNext;
           }
       }
       else
       {
           int bVoice  = Playlist_IsVoiceFile(pEntry);
           if((g_iPlaySet == PLAYSET_MUSIC && !bVoice) ||
              (g_iPlaySet == PLAYSET_VOICE &&  bVoice) ||
             (g_iPlaySet == PLAYSET_CUSTOM &&  !bVoice)   )
           {
               if(g_iPlaySet == PLAYSET_CUSTOM){
                 //the name is swizzled
         if((pEntry->m_pszFilename[0] & 0x0000FF) == 'A'){
               g_PlayList[g_iTotalTracks]= pEntry;
               g_iTotalTracks++;
       }
  }
               else{
```

```
                    g_PlayList[g_iTotalTracks]= pEntry;
                    g_iTotalTracks++;
        }
                }
            }

            rtn = PLAYLIST_SUCCESS;
        }
        return rtn;
    }
```

There are several pieces of code throughout the sdk, which check which PLAYSET is loaded before executing certain code. One such place is in the `SendPlayMode()` function. The custom play list from above will not allow for changing modes however it could if `SendPlayMode()` were changed to allow for it. It is encouraged that you search through the code for such places and tailor them to your specifications.

To show how the above changes have affected the player I have loaded 5 songs onto the player. Two of the songs have been renamed with SFNs that meet the criteria for being in the custom play list. The files are:

- 03-TRO~1.MP3 03 - Troubadour.mp3
- A2.mp3
- A1.mp3
- 21-ALI~1.MP3 21-Alison Krauss - The Lucky One.mp3
- BEACHB~1.MP3 Beach Boys - Help Me, Rhonda.mp3

Notice: The two files that have been renamed do not have Long File Names. This was done to stress that the algorithm used to add items to the custom play list is based on SFNs only. It would be possible to use long file names however this would require a bit more coding.

The screen shots on the following page is taken from the BoxView debugger:

The circled track names above correspond to tracks that will be included in the custom play list (‘A’ = 0x41). Note that they will also be in the music play list. This qualitative determination of files to be included in play lists can be quite powerful if used properly. The classification of files is not limited to the file name only. As can be seen from the sdk it may also depend on the type of file: wma, mp3, wav, etc. The play list may also be constructed based on what directory the file is in. Basically any data that can be gathered from a file can be used to classify it. Another type of classification could be based on file size for instance. While this implementation would require more extensive use of the SDK and its other APIs it is totally conceivable and with the above code should be realizable through minor modification.

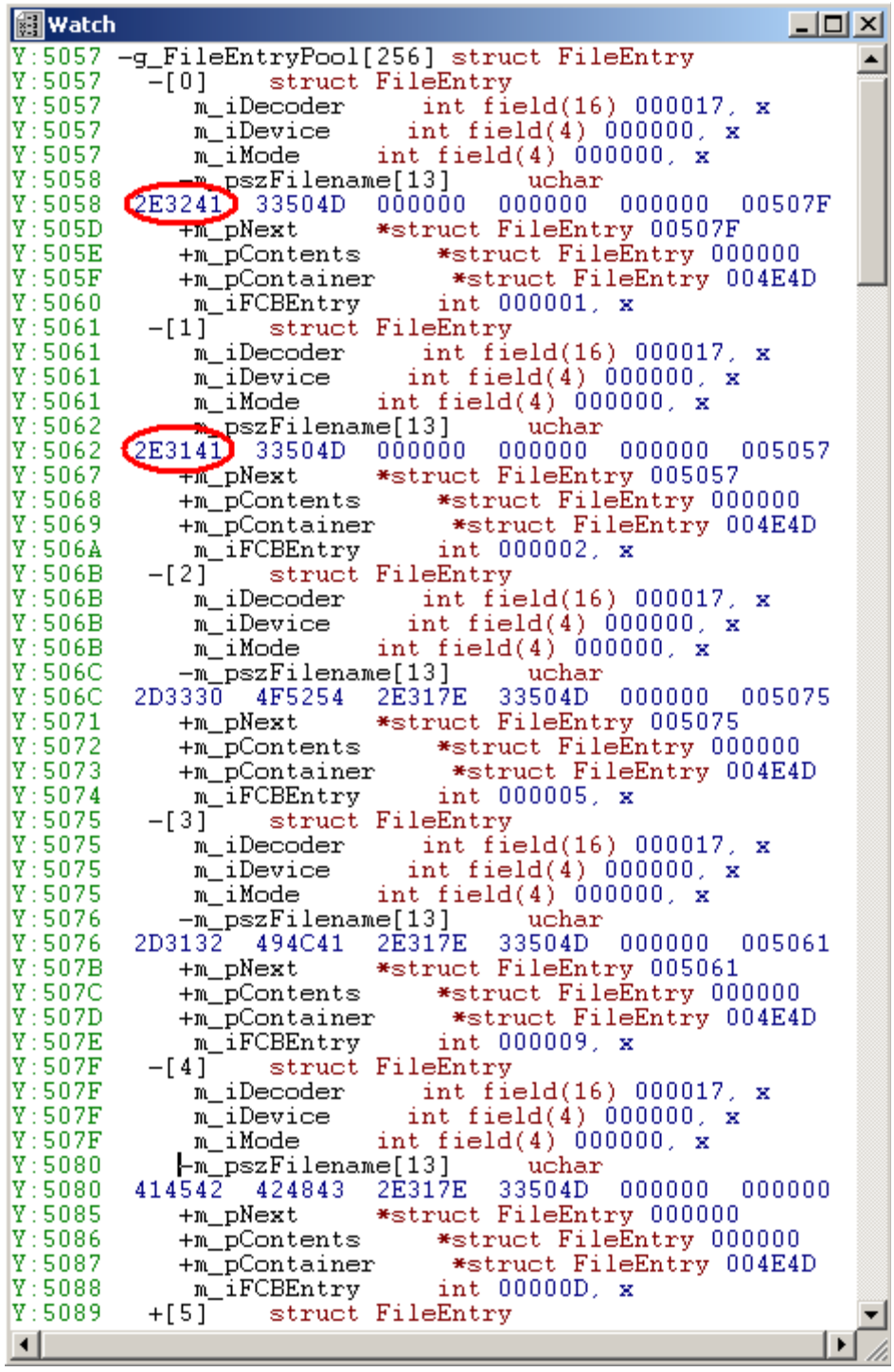

Watch
Y:5057 -g_FileEntryPool[256] struct FileEntry
Y:5057 -[0] struct FileEntry
Y:5057 m_iDecoder int field(16) 000017, x
Y:5057 m_iDevice int field(4) 000000, x
Y:5057 m_iMode int field(4) 000000, x
Y:5058 -m_pszFilename[13] uchar
Y:5058 2E3241 33504D 000000 000000 000000 00507F
Y:505D +m_pNext *struct FileEntry 00507F
Y:505E +m_pContents *struct FileEntry 000000
Y:505F +m_pContainer *struct FileEntry 004E4D
Y:5060 m_iFCBEntry int 000001, x
Y:5061 -[1] struct FileEntry
Y:5061 m_iDecoder int field(16) 000017, x
Y:5061 m_iDevice int field(4) 000000, x
Y:5061 m_iMode int field(4) 000000, x
Y:5062 -m_pszFilename[13] uchar
Y:5062 2E3141 33504D 000000 000000 000000 005057
Y:5067 +m_pNext *struct FileEntry 005057
Y:5068 +m_pContents *struct FileEntry 000000
Y:5069 +m_pContainer *struct FileEntry 004E4D
Y:506A m_iFCBEntry int 000002, x
Y:506B -[2] struct FileEntry
Y:506B m_iDecoder int field(16) 000017, x
Y:506B m_iDevice int field(4) 000000, x
Y:506B m_iMode int field(4) 000000, x
Y:506C -m_pszFilename[13] uchar
Y:506C 2D3330 4F5254 2E317E 33504D 000000 005075
Y:5071 +m_pNext *struct FileEntry 005075
Y:5072 +m_pContents *struct FileEntry 000000
Y:5073 +m_pContainer *struct FileEntry 004E4D
Y:5074 m_iFCBEntry int 000005, x
Y:5075 -[3] struct FileEntry
Y:5075 m_iDecoder int field(16) 000017, x
Y:5075 m_iDevice int field(4) 000000, x
Y:5075 m_iMode int field(4) 000000, x
Y:5076 -m_pszFilename[13] uchar
Y:5076 2D3132 494C41 2E317E 33504D 000000 005061
Y:507B +m_pNext *struct FileEntry 005061
Y:507C +m_pContents *struct FileEntry 000000
Y:507D +m_pContainer *struct FileEntry 004E4D
Y:507E m_iFCBEntry int 000009, x
Y:507F -[4] struct FileEntry
Y:507F m_iDecoder int field(16) 000017, x
Y:507F m_iDevice int field(4) 000000, x
Y:507F m_iMode int field(4) 000000, x
Y:5080 -m_pszFilename[13] uchar
Y:5080 414542 424843 2E317E 33504D 000000 000000
Y:5085 +m_pNext *struct FileEntry 000000
Y:5086 +m_pContents *struct FileEntry 000000
Y:5087 +m_pContainer *struct FileEntry 004E4D
Y:5088 m_iFCBEntry int 00000D, x
Y:5089 +[5] struct FileEntry

Figure 26.

13. METADATA AND STRING HANDLING

13.1. MetaData Support

13.1.1. MP3 ID3V2

- Current support for Artist, Song Title, Album Title.
- Support of ID3V2 tag before or after XING VBR header.
- Support of non-compressed, non-encrypted frames only.
- No support provided for ‘unsyncronization’.
- Big or little endian UTF-16 strings supported, with or without the byte order mark.
- Strings encoded as ISO-8559-1 are assumed to be DBCS encoded in whatever code page the firmware is built with.
- Support based on ID3V2v2.4.0 specifications.

13.1.2. MP3 ID3V1

- Current support for Artist, Song Title, Album Title.
- Strings are assumed to be DBCS encoded in whatever code page the firmware is built with.

13.1.3. WMA Metadata

- Current support for Artist, Song Title, Album Title.
- All strings are assumed to be 2 byte UNICODE

13.1.4. WAV Metadata

- Currently supported is the short file name that is used as the song title.

13.2. Metadata overview

The metadata string information is stored internally as UNICODE, with one character per DSP word. When required, the metadata retrieval functions will convert DBCS to UNICODE.

When the metadata retrieved by the playerlib, a central metadata function is called: GetFileMetaData(). This function takes the filename and a pointer to a metadata structure to fill out. The function then determines the extension of the filename and calls the appropriate metadata function: GetMp3MetaData(), GetWmaMetaData(), GetWavMetaData(). Each of these functions performs the necessary file lookups to fill out the metadata structure with the Artist, Title, Album, sample rate, song length, etc.

If the Title of the song is still 0 length after this step, the title is replaced with the short file name of the file. Any directory information is trimmed.

Finally, each of the strings (Artist, Album, Title) is trimmed of trailing spaces.

13.2.1. MP3 Metadata

ID3V1 and ID3V2 metadata information is supported. The metadata structure is first populated with the ID3V1 information, and then this information is overridden with whatever information is stored in the ID3V2 tag if available.

Our experience has shown that the ID3V1 tags are stored in ISO-8559-1 format, or Microsoft DBCS (CP950, CP932, etc) encoded for the Asian market. Because of this, the

ID3V1 metadata code assumes that the information is DBCS and will try to convert to UNICODE as part of the extraction routines. The ID3V1 metadata code also assumes that the tag is always the last 128 bytes of the file.

The ID3V2 framework is set up to handle the required parsing of the various elements of the ID3V2 tag. It finds an ID3V2 tag, verifies that it is formed correctly and consumes unwanted portions of the tag's header. It then steps through each frame and attempts to process it. Unknown frames are skipped over.

The frame identification method has been designed to better suit the STMP3410 memory architecture. Each frame identifier is 32 bits in the file, which we load the first 8 into a frame type, and the next 24 into a subtype. This seems to work well as the frame identifiers are organized with the first 8 bits being a general category (e.g. TIT1, TIT2, TALB, etc all start with T and contain text fields)

If a developer wants to add a new type of ID3V2 frame, then s/he should modify Id3V2ReadFrame() to process the new frame type, or add a new subtype to the 'T' block of code. It is highly recommended the code for this be put into a separate function (similar to the string functions) so that upgrades to the ID3V2 code can be performed without disturbing too much of the customized work.

The constants for the subframe types are defined in \system\common\id3v2\id3v2_internal.h and are in the typical dyslexic format. If looking at the file in a hex editor, and the bytes are 0x31 0x32 0x33("123", then when read into a DSP word, they will appear as 0x333231 ("321").

There are several data helper functions for extracting data from the ID3 TAG, that are commented in the source files:

- Id3V2ReadVariedStringToUnicodeBuffer()
- Id3V2ReadUTF16String()
- Id3V2ReadSyncSafeInt()

Additional documentation on ID3V2 tags: http://id3lib.sourceforge.net/id3/ (This may move around as the MPIAA has shut down the original website for ID3LIB at www.id3lib.org)

After the ID3V2 tag, there may be a XING VBR header, which we currently only extract the total frame count and calculate the total song size from. In the future, it would be possible to use the TOC get more accurate current time display on VBR MP3 files, but it has not been implemented.

Finally, the MP3 metadata code searches through the MP3 file, looking for 2 consecutive valid MP3 frames, using the bitrate from those. From there, it looks 100 frames ahead and verifies whether the file is VBR or not. If it is, it sets the VBR bit so the menu's can display that information if desired.

13.2.2. Get Current Song Type

There is global variable which will return the current song type, g_iSongtype, in file C:\your sdk directory\System\Common\metadata\metadata.c. Now four types of value have been defined:

```
#define UNKNOWN_TYPE   0
#define MP3_TYPE       1
#define WMA_TYPE       2
#define WAV_TYPE       3
```

COMPANY CONFIDENTIAL 9/26/03

13.2.3. Customize MP3 Metadata Display

There are three configurations to customize the LCD display, ID3V1 only, ID3V2 only, and ID3V1 or VID3V2. Please see table 14-16 below for details.

To change the metadata display, a user has to modify the file C:\your sdk directory\System\Common\metadata\mp3metadata.c. To assign the global variable, g_iTagSelected, with one of the following constants will configure the player to display different ways:

- NO_TAG_SELECTED (default)
- ID3V1_TAG_SELECTED
- ID3V2_TAG_SELECTED

	NO_TAG_SELECTED	Display	Note
The MP3 song file has the tag of	ID3V1	Title (V1), Artist(V1), and Album (V1)	
	ID3V2	Title (V2), Artist(V2), and Album (V2)	
	ID3V1 and ID3V2	Title (V2), Artist(V2), and Album (V2)	
	ID3V1 and ID3V2 mixed	Title(use V2 if V2 is available, otherwise use V1) Artist(use V2 if V2 is available, otherwise use V1) Album(use V2 if V2 is available, otherwise use V1)	File name if no title

Table 15. No tag selected (default).

	ID3V1_TAG_SELECTED	Display	Note
The MP3 song file has the tag of	ID3V1	Title (V1), Artist(V1), and Album (V1)	
	ID3V2	Filename	
	ID3V1 and ID3V2	Title (V1), Artist(V1), and Album (V1)	
	ID3V1 and ID3V2 mixed	Title(V1 if V1 is available, otherwise use filename) Artist(V1 if V1 is available, otherwise use NULL) Album(V1 if V1 is available, otherwise use NULL)	

Table 16. ID3V1 tag selected.

	ID3V2_TAG_SELECTED	Display	Note
The MP3 song file has the tag of	ID3V1	Filename	
	ID3V2	Title (V2), Artist(V2), and Album (V2)	
	ID3V1 and ID3V2	Title (V2), Artist(V2), and Album (V2)	
	ID3V1 and ID3V2 mixed	Title(V2 if V2 is available, otherwise use filename) Artist(V2 if V2 is available, otherwise use NULL) Album(V2 if V2 is available, otherwise use NULL)	

Table 17. ID3V2 tag selected.

13.2.4. WMA MetaData

A WMA file is windows media audio packets stored in an ASF transport. The WMA metadata framework is derived from the Advanced Systems Format Specification published by Microsoft, found at http://www.microsoft.com/windows/windowsmedia/WM7/format/asfspec11300e.asp.

The will search the WMA file for a valid object with the ASF_Header_Object GUID, then search within the header object for other objects that contain each individual piece of information. The Artist and Title are stored within the Content Description object, but the Album is stored within the Extended Content Description Object, which is a little trickier to get data from.

The Extended Content Description Object has multiple content descriptors, each having a name, a type, and a value. The descriptor that contains the Album data has a name of "WM/AlbumTitle". If a developer has a need to extract other data from the extended content descriptor s/he should follow the example in wmametadata.c where the album is extracted.

13.2.5. WAV Metadata

The WAV metadata code looks for a valid RIFF header at the beginning of the file, and grabs the file length and samples per second data, then calculates the song time from this information. It will return valid metadata information for non IMA-ADPCM files, but the ADPCM decoder will only support IMA-ADPCM.

13.3. String handling and Localization Overview

The string handling and display functions support both DBCS and UNICODE. Currently, only one DBCS code page can be supported per firmware build. There are generally two identical sets of functions and messages to manipulate and display these strings: (packed_)strlen, (packed_)strncpy, (packed_)strcpy, (packed_)strncmp, (packed_)strcmp, (packed_)strncat, and (packed_)strcat. For display messages, LCD_PRINT_STRING(_UNICODE)(_INV)_RSRC and LCD_PRINT_STRING(_UNICODE)(_INV)_ADDR are provided for displaying these strings.

There is currently no utility to create UNICODE string resources.

Additionally, there are functions available to convert from DBCS to UNICODE (but not the other way around as there is no unique mapping from UNICODE to many DBCS code pages), DBCStoUnicode().

Finally, there are functions for measuring the width of a string in pixels: GetTextWidthAddressUnicode, GetTextWidthResourceUnicode, GetTextWidthAddressDBCS, and GetTextWidthResourceDBCS. This function uses the current font selected to determine the width of the string.

UNICODE strings are stored one character per DSP word; DBCS strings are stored in a format where each character may take up one or two words and are packed tightly using every byte of the DSP word. The packed characters are stored in a "dyslexic" format where the first byte is in the lowest 8 bits of the word, the next in the middle 8, and the third in the upper 8 bits. When looking at the words in the debugger it looks very strange, but programmatically it makes more sense. (The first character is in the first 8 bits.)

Refer to the following examples of packed vs. unpacked strings. Each square represents a 24bit DSP word

The string “abcdef” is stored as follows in Unicode :

a	b	c	d	e	f	\0

Table 18. Unpacked String Example

The string “abcdef” is stored as follows in DBCS/packed format:

cba	fed	\0

Table 19. DBCS/Packed String Example

Also, the compiler will take packed strings and arrange them in a packed format that is backwards from the way the rest of the system uses packed data.

_packed char MyString[] = “abcdef”; will produce: :

abc	def	\0

Table 20. Compiler Packed String Example

Which will not work with the rest of the system. Additionally, the initialization data of ‘abc-def’ will be stored in a section named .yconst<filename> (where filename is the current C filename) and will be located in the default Y data section (Y_Extra). If enough of these strings are used, the Y section will fill up.

One way to avoid the const section is to initialize the data programmatically:

```
_packed char MyString[13];
…
((int*)MyString)[0] = 0x434241;     //ABC backwards
((int*)MyString)[1] = 0x464544;     //DEF backwards
((int*)MyString)[2] = 0;            //Terminating NULL
```

In this example MyString is cast to a pointer to an int, and then indexed off of that as indexes into packed pointers are undefined. This method does take up more program space than the previous method, but in most cases the program space is overlayed (and the .yconst<filename> cannot be).

13.4. Long File Names

Long File Names are a result of operating systems going beyond the original 8.3 naming convention for files. Both windows and mac long file names are supported by the Sigma-Tel SDK. Obtaining the Long File Name (LFN) is accomplished through either of two functions. One of the functions is used exclusively by the playlist API and the other is a general purpose function that will work with any file on the player. The long file name implementation used assumes that the LFNs are stored as unicode strings in the file system. This is true for Windows as well as Mac files.

13.4.1. Playlist_LFNGetFileName

RETCODE _reentrant Playlist_LFNGetFileName(int TrackNumber, int b,UCS3* LFNFile-Name)

Inputs: int TrackNumber - The track number for the LFN desired

UCS3*LFNFileName-pointertotheUCS3stringdestination(wheretheLFNis copied to)

Outputs: Length of the LFN string (0 for no LFN)

This function is SysCallFunction callable. Although it may be useful to call this function in your code it is automatically called during MetaData extraction for a file if there is no Meta-Data for a particular music file or voice file.

13.4.2. ChangeDirtoFileEntryDir

RETCODE _reentrant ChangeDirtoFileEntryDir(struct FileEntry *pCurrentEntry, _packed BYTE * buffer)

Inputs: struct FileEntry* pCurrentEntry - pointer to an LFN_FILE_LOCATOR

_packed BYTE *buffer - pointer to a buffer where the directory path can be safely constructed

Outputs: PLAYLIST_SUCCESS or PLAYLIST_END_OF_LIST

This function is a helper function for Playlist_LFNGetFileName although it may have other practical uses.

13.4.3. FSLFNGetFileName

RETCODE _reentrant FSLFNGetFileName(int fileHandle, int b, UCS3 *LFNFileName)

Inputs: int fileHandle - a filehandle for an open file. This is the file handle that is returned from an FSFileOpen call.

int b - present only to make the function SysCallFunction callable

UCS3* LFNFileName - the buffer where the LFN is to be copied.

Outputs: Length of the UNICODE string copied to LFNFileName array.

14. IO LIBRARY

The IO library is a collection of functions and routines that provide insight into the file system.

14.1. _find functions

The find functions provide a way to find file information when building a database of files available on the device (useful when creating playlists). These three functions, coupled with FSChangeDir() allow the developer to browse the file tree and identify files to include into the play queue.

The functionality is modeled after the Microsoft implementation of _findfirst, _findnext, _findclose, so the MSDN documentation can be used as a guide to use.

14.1.1. _findfirst()

_reentrant int _findfirst(_packed BYTE *filespec, struct _finddata_t *fileinfo)

This will start an inquiry, searching the current directory for short files that match the specification passed in. It will return the search handle, or –1 if no files are found or there are no free handles. This function does not check that fileinfo is properly filled out, or that the pointer passed in is valid.

14.1.2. _findnext()

_reentrant int _findnext (int handle, struct _finddata_t *fileinfo)

This will continue the search until the end of the directory is reached, returning one entry per call or –1 if the end of the directory has been found.

14.1.3. _findclose()

_reentrant int _findclose(int handle)

This function will terminate the search, freeing up the internal data structure required to perform the search.

14.2. Size Functions

14.2.1. GetDiskFreeSpace()

void GetDiskFreeSpace(void)

This function provides the total space in bytes remaining on a particular device.

15. CONTROLLING THE HARDWARE

15.1. Multi-stage Volume Control

15.1.1. Multi-Stage Volume Control Theory of Operation

The STMP3xxx mixer provides each input two volume control stages.

Master Volume Control : provides 31 steps of attenuation (46.5dB) plus mute all volume output

Individual Input Volume Control : provides 23 additional steps of attenuation (34.5dB) and 8 steps of gain (12dB) plus mute for each input -- these numbers match the datasheet, but actual attentuation/gain steps vary based on DAC 0dB point)

In the typical MP3 player, the DAC will be the primary mixer input source. The hardware design and PCM processing chain will determine which DAC gain setting corresponds to 0dB. The DAC volume level that corresponds to 0dB is defined as the maximum value that produces a non-distorted output when the Master volume is at 0dB. Typically, the 0dB DAC volume point will be in the 0x8 to 0xA range. The DAC 0dB point is adjustable in this architecture. The maximum Master volume setting is typically 0x0 but this can also be adjusted to allow for reduced headphone power output.

Many customers will find that it is desirable to provide some additional gain beyond 0dB. The 0dB point is defined as the maximum non-distorted value that will work on all silicon, across temperature and process. The majority of the silicon will also support an additional volume step. For example, if the master and DAC volume registers are set to their 0dB levels, the output may be 0.6V-peak since this is the maximum level supported at a Score of 1.55V. Adding a gain step so that the DAC volume is at +1.5dB will boost the output to 0.7v-peak, which will work without distortion on most silicon. Additional gain can be provided to allow for music that has been recorded with peak amplitudes below 0dB. The maximum DAC output boost is configurable in this volume control architecture.

Additionally, it is desirable to allow the PCM processing chain or other software to set a gain bias in the volume control. To ensure proper operation, the DAC data input cannot be allowed to exceed some value. The EQ or other PCM processing element will need to incorporate some auto-attenuation feature that prevents its output from exceeding this DAC input limit. The attenuation factor will always be a multiple of the volume step (1.5dB). The attenuation factor is shared with the volume control system. The volume control compensates for the attenuation by adding the same magnitude of gain in the mixer and master volume registers.

15.1.2. Multi-Stage Volume Control API

The multi-stage volume control allows the player statemachine to use a linear volume adjustment. This linear adjustment is mapped to values of the various volume control registers using the parameters defined for each project.

The SysVolume.c and SysVolume.h files, located in \system\common, contain the functions that map the linear user volume to the hardware registers (typically Master and DAC volume).

The SysUpdateVolume() routine adds the volume bias variable (Fg_VolumeBias) to the menu's target volume level, converts the linear volume level (0..64 in this example) to the proper register settings using the parameters in project.inc. It then sends messages to the mixer API to adjust the registers as necessary.

COMPANY CONFIDENTIAL 9/26/03

The SysIncrrementVolume() and SysDecrementVolume() functions simply increase or decrease the current linear volume level and call the SysUpdateVolume() routine.

The Fg_iUserVolume variable represents the linear volume set by the user. It is available globally by including SysVolume.h. It should be stored and retrieved on startup/shut-down along with other persistent system state variables.

The volume gain bias variable is shared by the volume adjustment code (which resides within the menu system) and the PCM processing code (which is part of the system). The Fg_VolumeBias variable is defined in sysmem.asm. It is prototyped in sysmem.h, which is typically included in the menus. The PCM processing chain and other assembly-based modules can access the variable through sysmem.xref.

The following volume adjustment parameters are stored in the project.inc file:

Parameter	Range	Typical	Description
MIX_DAC_NOM_VOL	0x0 .. 0x1F	0x6	DAC 0dB Volume level
MIX_DAC_MIN_VOL	MIX_DAC_NOM_VOL .. 0x1F	0x1F	DAC Minimum Volume setting
MIX_DAC_MAX_VOL	0x0 .. MIX_DAC_NOM_VOL	0x0	DAC Maximum Gain Setting
MIX_MSTR_MAX_VOL	0x0..0x1F	0x0	Master Maximum Volume (Used to limit the maximum Volume output)

Table 21. Multi-Stage Volume Project Specific Constants

The volume adjustment parameters (also in project.inc) are used to define several constants that are used within the system:

Equate	Equation	Comment
NUM_DAC_ATT_STEPS	MIX_DAC_MIN_VOL - MIX_DAC_NOM_VOL + 1	Number of steps in the DAC attenuate stage - includes DAC at NOM.
NUM_MSTR_ATT_STEPS	0x1F - MIX_MSTR_MAX_VOL + 1	Number of steps in the master volume attenuate stage.
NUM_DAC_GAIN_STEPS	MIX_DAC_NOM_VOL - MIX_DAC_MAX_VOL + 1	Number of steps in the DAC volume gain stage.
NUM_DAC_VOL_STEPS	NUM_DAC_ATT_STEPS+ NUM_MSTR_ATT_STEPS+ NUM_DAC_GAIN_STEPS+1	Total number of steps in the volume control (+1 is because of mute, which is always volume 0).

Table 22. Multi-Stage Volume Definitions

The volume control configuration defined above provides 4 stages:

Stage	Volume Range	Master Vol. Range	DAC Vol. Range	Mute	Steps in stage
0	0	0x1F	0X1F	1	1
1	1..26	0X1F	0X1F..0X6	0	26
2	27..58	0X1E..0X0	0X6	0	32
3	59..64	0X0	0X5..0X0	0	6

Table 23. Volume Stages

Stage 0 is simply mute.

Stage 1 is the DAC volume attenuation stage. The master volume is held at maximum attenuation and the DAC volume is adjusted through its attenuation range. This stage ends when the DAC gain is at its 0dB value.

Stage 2 is the master attenuation stage. The DAC volume is held at its 0dB level (0x9 in this example) and the master volume is adjusted. This stage ends when the master volume reaches its 0dB level (level of least attenuation).

Stage 3 is the DAC gain stage. The Master volume is held at its maximum level (0 in this case) and the DAC gain is increased to its maximum value (0x0 in this case).

*Please note that in the STMP volume control registers (as in AC97), higher register values correspond to higher attenuation or lower gain.

15.1.3. Reading the current volume level

The volume level display uses the g_iUserVolume global variable to determine the current user volume level. It also uses the g_iVolSteps from SysVolume to determine the total volume range. The total range depends on the current mixer volume and mixer mode. For example, in MP3 playback there may be 64 total volume steps while in FM mode, the maximum volume may be limited so that there are only 60 steps.

The volume indicator code is responsible for mapping the linear volume level and range to the actual LCD. In many systems, the total number of steps will exceed the number of indicator icons. In these cases, more than one volume step may pass before the indicator is changed. It is also possible to reduce the number of steps, either by doing multiple volume increments at a time in the PlayerStateMachine.c or adjusting the multi-stage volume declarations in project.inc. An example of the latter would be to increase the MIX_DAC_MAX_VOL to remove some of the volume gain steps, or decrease the MIX_DAC_MIN_VOL to remove some of the lower-volume steps above mute.

15.2. PLL/ Voltage

SysSetSpeed.asm along with Sysspeed.inc provide several levels of speed an voltage settings to optimize system performance.

Note: SigmaTel does not recommend changing this algorithm or speed/voltage settings. A power management API will be provided in a future SDK that will allow the menu level to have better control over the system speed.

15.2.1. Basic Algorithm For Adjusting PLL

- when increasing frequency, increase the voltage first
- when decreasing frequency, decrease the frequency first
- when adjusting voltage the brownout voltage should be considered:
  - lower the brownout voltage before the core voltage is lowered
  - raise the core voltage and settle before setting brownout voltage level

15.2.2. User Level Requirements For PLL Control

- Initialize IDLE speed after reset so the algorithm will follow the case of decreasing frequency as the first setting after reset. (Reset values are : crystal 24.576MHz / Vddd = 1.77)
- Do not modify speed indexes (SPEED_MP3, SPEED_IDLE, etc.) because the value is dependent upon the order of the speed table in SysSetSpeed.asm.
- CAUTION: SPEED_MAX is lower than SPEED_WMA. SigmaTel to fix the naming in a future SDK release.

15.3. Battery Monitoring

Battery level monitoring for the 3410 uses the 7 bit LRADC.

Note: **YOU MUST HAVE A TA3 PART! If you do not have a TA3T100 the LRADC does not function properly. The older engineering boards typically have TA2 parts.

15.3.1. Rechargeable LION

SDK2.320 does not include the firmware requirements needed to incorporate a rechargeable LION battery. Please see the application note on LION batteries, which describes the firmware requirements.

15.3.2. LRADC Sampling API

LRADC sampling is combined with the button isr in the file button.asm. The raw sample is stored as well as an average of every X samples for all powered channels. The averages are also computed in the isr.

Initialize each LRADC channel as disabled/enabled and with selected reference. When the result register reads 0x7f, the voltage equals the selected reference.

```
SysLowResolutionADCInit (channel,power,reference);
```

System routine that will return raw or averaged voltages in mV based on the reference voltage selected during initialization.

```
INT SysLowResolutionAdcReadVolt(BYTE bAdcChannel, BOOL bAv-
erage);
```

System macros returning current average or last value read:

```
SysLowResolutionAdcReadAvg (channel)
SysLowResolutionAdcReadBin (channel)
```

Controls the number of samples per average (defined in button.asm)

```
LOW_RES_ADC_AVG_CNT      equ 1<<LOW_RES_ADC_AVG_SHIFTER
```

In SDK examples, the isr occurs every 40mS and there are 256 samples in an average. Therefore the new average is updated every 10.24 seconds.

15.3.3. Battery Voltage Reporting API

Returns % of battery level based on range defined in project.inc.

```
SysBatteryGetLevel (channel)
```

where,

VBATT_MIN_MV is the bottom of battery valid range in mV and represents 0% battery capacity.

VBATT_MAX_MV is the top of battery valid range in mV and represents 100% battery capacity

15.3.4. SDK Battery Monitoring Example Implementation

Use a softtimer to send menu message (MENU_BATTERY_CHNG) every 2 seconds to control updating the display. There is some latency in the current implementation which can be controlled by the softtimer interval and the number of sample in the average.

- Initialize LRADC and softtimer (lcdexample - mainmenu.c)

```
SysLowResolutionADCInit (BATT, OFF,VBATT_REF);
SysPostMes-
sage(6,SOFT_TIMER_SET_TIMER,TIMER_BATT_CHK,0,2000,MENU_BATTE
RY_CHNG);
```

- When the message is received the battery display flag is set and refresh display flag is set. (all menus service this message).

```
case MENU_BATTERY_CHNG:
  bRefreshDisplay =TRUE;
  DisplayHint.bits.Battery = TRUE;
  break;
```

- In the display function the integer percentage of the battery level remaining is returned. The range used in calculating the battery life % is defined in project.inc (currently set for 1.5V - .9V)

```
SysBatteryGetLevel (channel)
```

15.3.5. USB Battery Monitoring (USBMSC)

Since LRADC sampling is included within the button isr, a new function, GetBatteryLevel(), was created to handle battery detection for the USBMSC firmware. This function will perform the following actions:

- Read current LRADC value
- Set Low Battery Flag if below threshold
- Calculate battery capacity
- Return the battery level based on the available resources (See **Section 20.2 USB Battery Display** for more information)

Note: The USBMSC does not use the global *project.inc* and has a local version within its folder (...\lcdexample\usbmsc), which requires the battery defines to be assigned in two locations.

15.4. ADC Output

ADC ISR -- ADC is controlled as part of the encoder module. Configuration is setup before recording.

15.5. DAC Output

DAC ISR -- DAC is controlled and configured as part of the decoder module.

16. PLAYER SHUTDOWN

16.1. Shutdown Events

A special menu is setup in SDK examples for shutting down the player. (? NAME?) The parameter passed in will either shutdown immediately or will display to the user the shutdown menu and wait until the STOP button is pressed for X seconds.

SDK examples shutdown by posting the message SYSTEM_SHUTDOWN_FINAL under the following conditions: autoshutdown, user shutdown, usb connected.

16.1.1. Timed Auto Shutdown

The Settings menu allows the player to shutdown after a selected time of no activity. A softtimer (name) is programmed to expire after X minutes. Everytime a button event is processed or the decoder/encoder is started the timer is restarted. When it expires a shutdown message is sent to the menus and the player is shutdown.

IMPT! If this feature does not work make sure the Tasking compiler patch is installed from the SigmaTel Extranet.

16.1.2. User Shutdown

The SDK examples require the user to hold the STOP button for X in order to shutdown the player. This can occur in almost every menu so the following points are important when reusing this technique.

Configure X using the constant SHUTDOWN_COUNT defined in mainmenu.c

When SHUTDOWN_COUNT equals 1, shutdown will occur in ~620mS. Each count after that increments the time by ~320mS. The example defines SHUTDOWN_COUNT = 6 enabling shutdown after the STOP button is pressed for ~2.2 seconds.

The current solution in the examples are limited to SHUTDOWN_COUNT range of (1 to 6) without changes to the display function.

The routine can be called from any codebank so use the prototype that SysCallFunction supports.

```
_reentrant int ShutdownMenu( int bForceShutdown, int bLow-
Batt, int *pPtr)
```

16.1.3. Exception due to Brownout or Headphone Short

When special cases occur, the player is shutdown differently. Typically the player is shutdown in the routine processing the exception and does not use the system module.

16.2. Save Settings

SysSaveSettings() is called from the system module when the SYSTEM_SHUTDOWN_FINAL message is processed. Shutdowns due to brownout or headphone short will not save settings.

Note: IMPORTANT -- The current SDK architecture causes the display to be cleared indicating player shutdown approximately 1 second before the player is safe to remove power. This has caused flash corruption problems in manufacturing when power supplies are removed after shutdown while the flash is written. SigmaTel plans to address this issue in a future SDK and is referenced in the Known Defect list as defect #Stmp00002902.

COMPANY CONFIDENTIAL 9/26/03

16.2.1. Battery Level Dependency

SysSaveSettings() checks the battery level before writing settings and will not write settings to flash if the battery level is below as certain threshold (BATT_SAFETY_MARGIN defined in project.inc).

BATT_SAFETY_MARGIN represents the required % of battery life remaining before shutting down. `SysBatteryGetLevel()` is used to determine the battery life.

SDK examples use 10%, so assuming the battery range is .9-1.5V the settings will only be saved at battery levels above 0.96V.

16.2.2. Saving Variables

Use the assembly macro SaveRange to save any global variable during the call to SysSaveSettings().

The macro must be called from an assembly file or inline assembly, so the assembly symbol name is used. The macro must reside in section `Player_settings_Y.` The format of the macro is:

`SaveRange` *`symbolname #words`*

The code related to the macro will reside in a section that is defined outside of the usable memory.

Below is an example from the SDK eqmenu.c that will save 1 word from memory location defined by the variable g_iEqSetting.

```
#pragma asm
    include "sysmacro.asm"
    org y,"Player_settings_Y":
    extern y:Fg_iEqSetting
    SaveRange Fg_iEqSetting,1
#pragma endasm
```

16.2.3. Settings.dat format

Settings.dat is a binary file written to the NAND flash containing the values of global variables. When the settings are restored the memory locations stored in this file are overwritten. Be careful to provide default values for cases when settings.dat is not available.

If the saved version does not match the firmware version running, the settings will not be restored.

The same memory location pointers as used by the filesystem complex pointers are used here. Xmemory = 0x80, Y memory = 0x40, P memory = 0x20

VERSION MAJOR, VERSION MINOR, absolute address, size in bytes, value

Example:

```
020000VERSION MAJOR 2
690000VERSION MINOR 105
304280 x:0x4230
030000 3 bytes (1 word)
0B0B00 0x0B0B value
7A0040 y:0x007A
030000 1 word
000000 0x0000 value
```

In the above example the following memory locations will be overwritten when the settings are restored:

x:0x4230 = 0x0b0b

y:0x007A = 0

16.3. Headphone Shutdown

The same approach as power on has been added to the SDKs. The DAC signal is ramped slowly to ground before powering off headphone amp.

control variables:

INITIAL_DAC_VALUE -- used when

16.4. RTC Shutdown

Chip errata -- Due to a defect in the STMP3410, a special RTC shutdown must occur before the CKRST bit is set or the RTC value will be corrupted. Refer to the SysRTCShutdown() routine in sysbrownout.asm for details.

ROM code does not perform the correct workaround during POST operation and will corrupt the RTC is POST is enabled. POST operation is controlled via bootmode pins.

Due to the extended time required to shutdown the RTC correctly (typically 500uS but could be up to 1.5mS), this shutdown method is not used during the Battery Fallout brownout case.

Refer to the RTC ECN on extranet.

16.5. Power Off

The final step in shutting down the player is disabling the DC/DC converter.

```
bset    #HW_CCR_PWDN_BITPOS,x:HW_CCR
```

COMPANY CONFIDENTIAL 9/26/03

17. SYSTEM MESSAGING MODULES

Besides the modules described in the User Interface section of this document, other messaging modules are accessed by the menus to control player operation. A description of the remaining modules is provided in this section.

17.1. Soft Timers (stmp3xxx_sdk\system\msgmodules\software\softtimer\softtimers.c)

The softtimer module provides single and reoccurring monotonic time-out events by posting module messages (generally to the menus).

The number of available timers in a system is defined by a constant in project.inc. Each timer requires 5 words of Y memory and the number of sofftimers can affect response time.

```
; Number of active timers
SOFT_TIMERS  equ 3
```

The sofftimer module will only post a message without parameters and does not perform error checking on the message.

```
SysPostMessage(2,pTimer->m_iMsg);
```

Proper consumption of the messages is the responsibility of the target module. (i.e. If a menu configures a soft timer to post a message in 10 minutes, and the user has moved to another menu, that message must at least be consumed and destroyed).

There is no timer arbitration. For example, if softtimer 0 is set from multiple places in the system, the timer API will allow the last message to reconfigure the timer.

17.1.1. Softtimer Message Definitions

# words	Message Name	Arguments	Message Description
6	SOFT_TIMER_SET_TIMER	Timer ID Count Timeout Message to Post	0..MAX_TIMERS-1 Number of times message will reoccur (23 bits), where 0 is infinite count In milliseconds (48 bits) A message with no parameters can be posted to any module when timer expires
3	SOFT_TIMER_KILL_TIMER	Timer ID	0..MAX_TIMERS-1

Table 24. Softtimer Messages

17.1.2. Example Softtimer Messages

Setup timer to send reoccurring animation message to menus that can be used to control scrolling the display -- every 200mS the `MENU_MSG_ANIMATE` message will be posted.

```
SysPostMes-
sage(6,SOFT_TIMER_SET_TIMER,TIMER_ANIMATE,0,200,MENU_MSG_ANI
MATE);
```

Disable timer when go to another menu so don't have to needlessly process animation messages. If do not kill the timer, the new menu must process or ignore the `MENU_MSG_ANIMATE` message.

```
SysPostMessage(3,SOFT_TIMER_KILL_TIMER,TIMER_ANIMATE);
```

17.2. Parsers

SDK2.2xx Note: This module has been incorporated into the project-specific menu codebank architecture in SDK2.2xx. MENUS SERVICE THESE MESSAGES NOW

.

#words	Message Name	Arguments	Message Description
	PARSER_NEXT_SONG	0 = NextSong 1 = NextSong + EOF 2 = NextSong + Button 3 = NextSong + Ffwd	Parser finds the next song based on the playmode and playstate. The song information will be extracted during this operation. '0' loads the next song with no further action '1' loads the next song and posts a DecoderPlay message IF the current song was NOT the last song. '3' loads the next song AND posts a Decoder Play message. '4' loads the next song and post a Ffwd message.
	PARSER_PREV_SONG	0 = PrevSong 1 = PrevSong + Play 2 = PrevSong + Rwnd	Parser finds the previous song in the list or directory based on the playmode and playstate. The song information will be extracted during this operation. '0' loads the previous song with no further action '1' loads the previous song and posts a Decoder Play message '2' loads the previous song and posts a Rwnd message to the decoder.
	PARSER_STOP	None.	This message will cause the parser to reset the pointers to the beginning of the play list or directory.

Table 25. Parser messages initiated by the decoder

#words	Message Name	Arguments	Message Description
	PARSER_DEVICE_ENUMERATE	None.	This message is sent when a device (card) has been removed from (or inserted into) the player.If the decoder is playing, the parser will first send a message to Stop the decoder and another device enumerate message back to the parser.If not playing, the parser will recount the songs and reset to the first song.
	PARSER_SET_CURRENT_SONG	Song Number	Parser will advance playlist to the input track number. File information will be extracted but will not automatically play.

Table 26. Parser messages called from menu level

17.3. Decoders

All decoder modules support the same message API. If the decoder does not use the message it can be ignored, but must still be processed by the module. Most of the functionality is identical between modules. The message definitions below only reference the MP3 decoder.

17.3.1. Decoder Message Definitions

Message Name	Arguments	Message Description
DECODER_RESET	None	Clear DecoderIgnoreMessages flag
DECODER_PLAY	None	post MENU_SONG_TIME_CHNG If playing, pause the decoder: Fade out to mute Stop the DAC Post message: MENU_DECODER_STATE_CHG If stopped or paused, begin playback: Adjust pll to SPEED_MAX (allow faster sync) DecoderOpenSong() Post message: MENU_SONG_TIME_CHNG Init DAC Init MP3 decoder Enable Dac Start DAC ISR timer Post message: MENU_DECODER_STATE_CHG Wait for file sync. Other modules can run. If timeout assume badfile and skip current track. If bitrate < 256 kbps, adjust pll to SPEED_MP3 Fade in mixer from mute. Adjust status flags.
DECODER_STOP	None	StopCurrentSong() DecoderCloseSong() Clear decoder source and sink buffers. DecABModeDisable() Adjust playback status flags. Free DCLK for other speed clients Post messages: PARSER_STOP MENU_DECODER_STATE_CHG MENU_SONG_TIME_CHNG

Table 27. Decoder Messages

Message Name	Arguments	Message Description
DECODER_FFWD	None	DecoderOpenSong() post message: MENU_DECODER_STATE_CHG If the end of file is reached before the number of bytes to skip the next song is loaded: - DecABModeDisable() - StopCurrentSong() - DecoderCloseSong() - Free DCLK for other speed clients - post messages: MENU_DECODER_STATE_CHG PARSER_NEXT_SONG w/ Ffwd (Menus will now handle parser messages) - playerstatemachine posts another DECODER_FFWD message when the next song is loaded. - Set DecoderIgnoreMessages flag If there are enough bytes in file to seek: - InitInputBuffer() - CopyBufferInit() - Mp3DecInit() - DacDriverInit() - DecoderFileSeek() - post message: MENU_SONG_TIME_CHNG
DECODER_RWND	None	If the beginning of the file is reached before the number of bytes to skip the previous song is loaded: - DecABModeDisable() - StopCurrentSong() - DecoderCloseSong() - Post messages : MENU_DECODER_STATE_CHG PARSER_PREV_SONG w/ Rwnd (menu will now handle) - playerstatemachine posts another DECODER_RWND message after the next song is loaded. - Free DCLK for other speed clients - Set DecoderIgnoreMessages flag. If there are enough bytes in file to seek: - DecoderOpenSong() - InitInputBuffer() - CopyBufferInit() - Mp3DecInit() - DecoderFileSeek() - Post message: MENU_SONG_TIME_CHNG MENU_DECODER_STATE_CHG

Table 27. Decoder Messages (Continued)

Message Name	Arguments	Message Description
DECODER_NEXT_SONG	None.	Update status flags. DecABModeDisable() If stopped: DecoderCloseSong() Post messages: PARSER_NEXT_SONG (menu will handle) MENU_DECODER_STATE_CHG Set DecoderIgnoreMessages flag Free DCLK for other speed clients If playing or paused: StopCurrentSong() DecoderCloseSong(); Post messages: MENU_DECODER_STATE_CHG PARSER_NEXT_SONG w/ Play --- menu will process and post a DECODER_PLAY message after the next song is loaded. Set DecoderIgnoreMessages flag Free DCLK for other speed clients
DECODER_PREV_SONG	None.	Update status flags. DecABModeDisable() If stopped: Post messages: PARSER_PREV_SONG MENU_DECODER_STATE_CHG Set DecoderIgnoreMessages flag Free DCLK for other speed clients If playing or paused: StopCurrentSong() DecoderCloseSong() Post messages: MENU_DECODER_STATE_CHG PARSER_PREV_SONG w/ Play --- menu will process and post a DECODER_PLAY message after the previous song is loaded.
DECODER_TIME_MODE	None.	Toggles between decoder posting song time as "elapsed time" (default) or as "remaining time". Toggles TimeMode flag in Fg_wDecoderCSR

Table 27. Decoder Messages (Continued)

COMPANY CONFIDENTIAL 9/26/03

Message Name	Arguments	Message Description
DECODER_AB_MODE	None.	Marks position of song for AB looping. First time received: A position is set. File location is stored and AB_Mode_A bit is set in Fg_wDecoderCSR Second time received: B position is set. File location is stored and AB_Mode_B bit is set in Fg_wDecoderCSR. If the A position is not less than the B position, AB mode will be disabled. Decoder will then loop between these points. Third time received: AB mode is disabled

Table 27. Decoder Messages (Continued)

17.3.2. Decoder Functions Specific To Each Module

Only MP3 decoder functionality is listed. Refer the decoder module overlay code for exact functionality of each module.

_asmfunc void StopCurrentSong(void);

- Stops playback and fades out mixer to muted.
- Updates Fg_wDecoderCSR playstate status flags.
- Stop DAC transmission

_asmfunc void DecABModeDisable(void);

- Clears AB status flags in Fg_wDecoderCSR
- Clears file location markers

_asmfunc void DecoderCloseSong(void);

- FILEClose if not already closed
- Fg_wDecoderCSR: clear FileIsOpen flag

_asmfunc void DecoderOpenSong(void);

- FILEOpen if not already opened
- Fg_wDecoderCSR : set FileIsOpen flag

_asmfunc ccr_error DecoderFileSeek();

- Performs FileSeek
- Updates track time info

17.3.3. Common Decoder Functions Used By Decoder Modules

_asmfunc void InitInputBuffer();

- Initializes head and tail pointers to the beginning of the buffer.

void CopyBufferInit(int BufDescrPtr,int MemoryTypeFlag);

- Sets the decoder descriptor and memory type
- Clears the copy sink buffer

_asmfunc void DacDriverInit(void)

- initializes the dac buffers, sets up the hardware properly

_asmfunc void Mp3DecInit(void);

- Initialization Routine for MP3 Decoder algorithm. Initializes variables, sets up buffers. Must be called for each new song to restart sync.

17.3.4. DecoderSR / DecoderCSR Flag Definition

Fg_wDecoderSR:

Fg_wDecoderCSR:

Bit Name	Bit #	Description
PlayList	0	
PlayDir	1	
RepeatSong	2	
RepeatAll	3	
Random	4	
Pause	5	
Stop	6	
Rwnd	7	
Ffwd	8	
TimeMode	9	
DecSync	10	set when Fstatus_0 = 1 for MP3
Play	12	
EndOfSong	13	
EndOfList	14	
SongInfo	15	
FileIsOpen	16	set when a song file is currently opened by the decoder
SkipBlockHeader	17	used by parser
ABMode_A	18	
ABMode_B	19	
ABQuiet	20	
BadFile	21	set if bad file encountered
SyncWait	22	mp3 decoder looking for sync
FileReadError	23	error reading file

Table 28. DecoderSR/DecoderCSR Bits Parser

17.4. Encoders

Currently only a single encoder module is provided

17.4.1. Encoder Message Definitions.

Message Name	Arguments	Message Description
ENCODER_RECORD	None	Starts the EncoderModule. This message causes a file to be created and recording to begin. This message loads the encoder application overlay into memory.
ENCODER_STOP	None	Stops the EncoderModule. This message cause recording to stop, the new file to be updated with relevant header information, then closed.

Table 29. Encoder Messages

17.4.2. EncoderSR / EncoderCSR Bits

Bit Name	Bit #	Description
EncNotEnoughSpace	0	
EncRanOutOfSpace	1	
EncRecordError	2	
EncToldToStop	3	
EncAlreadyRecording	4	
Stop	6	
Play	12	

Table 30. EncoderSR/EncoderCSR Bits Parser

17.5. Mixer

Message module that controls the mixer in STMP34xx by abstracting the register setup to perform certain mixing functions.

17.5.1. Mixer Message Definitions

Message Name	Arguments	Message Description	STMP REGISTER
MIXER_MASTER_INCR	None	Increment master out level by one step	HW_MIXMASTERVR
MIXER_MASTER_DECR	None	Decrement master out level by one step	HW_MIXMASTERVR
MIXER_MASTER_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is muted	Set master out level to specified value	HW_MIXMASTERVR
MIXER_MASTER_MUTE	None	Mute the master output	HW_MIXMASTERVR
MIXER_MASTER_UNMUTE	None	Unmute the master output	HW_MIXMASTERVR

Table 31. Mixer Messages

COMPANY CONFIDENTIAL 9/26/03

Message Name	Arguments	Message Description	STMP REGISTER
MIXER_MASTER_FADE_OUT	None	Fades the master output to silence	HW_MIXMASTERVR
MIXER_MASTER_FADE_IN	None	Fades the master input to the current volume level	HW_MIXMASTERVR
MIXER_MASTER_BAL_RIGHT	None	Makes the right channel one step louder relative to the left channel	HW_MIXMASTERVR
MIXER_MASTER_BAL_LEFT	None	Makes the left channel one step louder relative to the right channel	HW_MIXMASTERVR
MIXER_MIC_INCR	None	Increase mic in level by one step	HW_MIXMICINVR
MIXER_MIC_DECR	None	Decrease mic in level by one step	HW_MIXMICINVR
MIXER_MIC_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is muted	Set mic in level to specified value	HW_MIXMICINVR
MIXER_MIC_MUTE	None	Mute the mic input	HW_MIXMICINVR
MIXER_MIC_UNMUTE	None	Unmute the mic input	HW_MIXMICINVR
MIXER_MIC_BOOST	None	Increase the mic in level by 20 dB	HW_MIXMICINVR
MIXER_MIC_UNBOOST	None	Decrease the mic in level by 20 dB	HW_MIXMICINVR
MIXER_LINE_INCR	None	Increase line in level by one step	HW_MIXLINE1INVR
MIXER_LINE_DECR	None	Decrease line in level by one step	HW_MIXLINE1INVR
MIXER_LINE_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is muted	Set line in level to specified value	HW_MIXLINE1INVR
MIXER_LINE_MUTE	None	Mute the line input	HW_MIXLINE1INVR
MIXER_LINE_UNMUTE	None	Unmute the line input	HW_MIXLINE1INVR
MIXER_FM_INCR	None	Increase FM in level by one step	HW_MIXLINE2INVR
MIXER_FM_DECR	None	Decrease FM in level by one step	HW_MIXLINE2INVR
MIXER_FM_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is muted	Set FM in level to specified value	HW_MIXLINE2INVR
MIXER_FM_MUTE	None	Mute the FM input	HW_MIXLINE2INVR
MIXER_FM_UNMUTE	None	Unmute the FM input	HW_MIXLINE2INVR

Table 31. Mixer Messages (Continued)

COMPANY CONFIDENTIAL 9/26/03

Message Name	Arguments	Message Description	STMP REGISTER
MIXER_DAC_INCR	None	Increase DAC In level by one step	HW_MIXDACINVR
MIXER_DAC_DECR	None	Decrease DAC In level by one step	HW_MIXDACINVR
MIXER_DAC_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is muted	Set DAC In level to specified value	HW_MIXDACINVR
MIXER_DAC_MUTE	None	Mute the dac input	HW_MIXDACINVR
MIXER_DAC_UNMUTE	None	Unmute the dac input	HW_MIXDACINVR
MIXER_ADC_SELECT	Source: 0x0000000 MicIn 0x000303 FmIn 0x000404 LineIn 0x000505 MixOut	Selects the input record source	HW_MIXRECSELR
MIXER_ADC_INCR	None	Increases the input gain of the record source	HW_MIXADCGAINR
MIXER_ADC_DECR	None	Decreases the input gain of the record source	HW_MIXADCGAINR
MIXER_ADC_SETLVL	Level to set 0x000000-0x000020 0x000000 is max 0x000020 is mute		HW_MIXADCGAINR
MIXER_ADC_MUTE	None	Mutes the input record source gain.	HW_MIXADCGAINR
MIXER_ADC_UNMUTE	None	Unmutes the input record source gain.	HW_MIXADCGAINR

Table 31. Mixer Messages (Continued)

COMPANY CONFIDENTIAL 9/26/03

18. FOREGROUND PROCESSING CHAIN

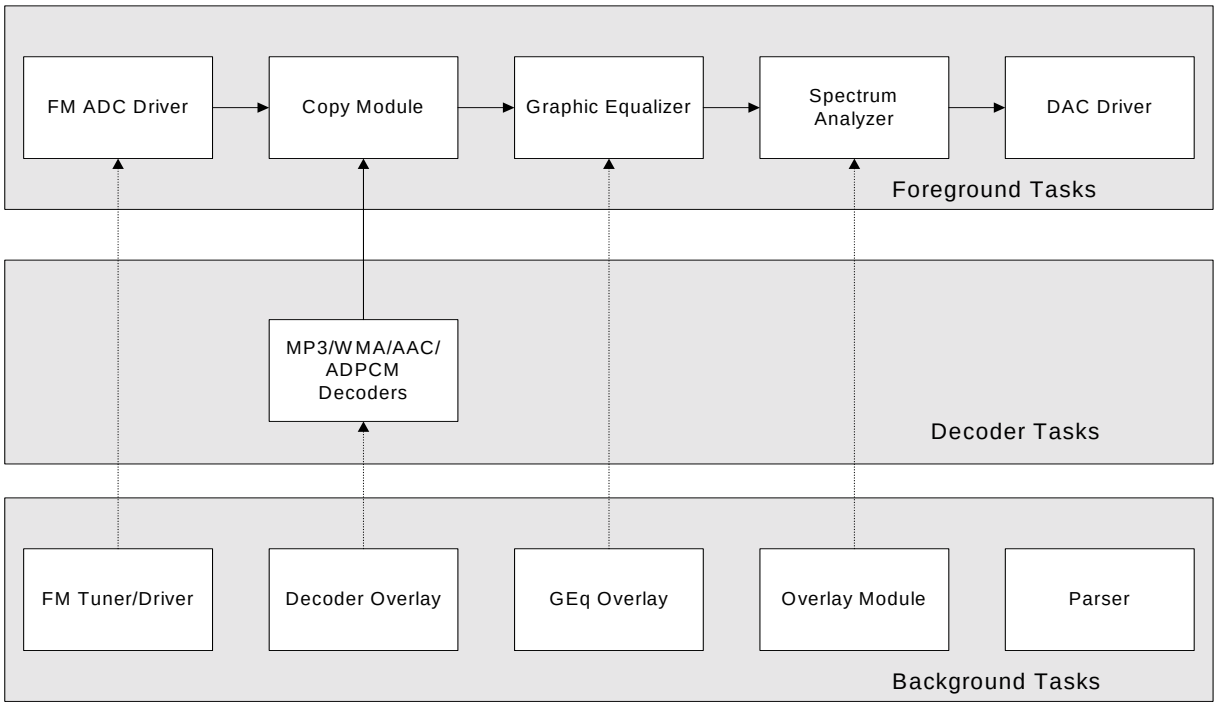


Figure 27. Foreground Processing Chain

19. FILE SYSTEM / MEDIA

19.1. Multi-Nand

Allows multiple NAND chips to be concatenated together to represent a single drive from the user perspective. If there is no external drive, the host can be customized to only display a single drive. SDKs do not support multiple NANDs as separate host drives anymore.

19.1.1. Hardware requirements:

Multiple NAND chips MUST have consecutive chip selects starting with CE0#.

NAND only -- up to 4 NANDs CE0#-CE3#

NAND and external SM -- up to 3 NANDs CE0#-CE2# ; external SM CE3#

NAND and external MMC -- up to 4 NANDs CE0#-CE3# ; external MMC any available GPIO (CE3# is default in software)

Note: In current SDK examples and reference designs, the MMC chip select uses CE3#. If this is changed in hardware, change the MMC configuration structure MmcPhysicalBus[] in mmchalstructs.c

19.1.2. Build option

Change the constant in project.inc to represent the number of NANDs in the system.

```
SM_INTERNAL_CHIPS            equ     1
```

19.1.3. Initial Configuration and Setup

1. Rebuild BOTH the usbmsc and player with the constant changed to define the number of NANDs in the system.
2. Verify the hardware connections are correct.
3. Copy the new firmware binary files to the host (\ProgramFiles\SigmaTel\Stmp34xx\).
4. USE RECOVERY MODE to enumerate the drives.
5. Format the NAND drive. Notice that Windows will report the size of only the first NAND until the SIGMATEL FORMATTER has been run on the drive. After the format is complete, the sum of all the NANDs will be the reported size.
6. Update the Firmware.

19.2. File System

19.2.1. FileSystem API

Will be updated with new filesystem introduction in SDK2.420.

19.2.2. FAT

Only FAT12/FAT16 formatting supported. FAT32 coming in SDK2.420.

20. USBMSC

20.1. DISPLAY CUSTOMIZATION

SDK 2.400 has made the transition from ASM to C for all of the display functions in the USBMSC firmware. The major change is using the C wrapper for SysPostMessage as is done in the menus.

Information on the usage of the SysPostMessage function can be found in **Customizing Flash Players Users Guide** available from the SigmaTel extranet website. A quick synopsis of the usage of the function for displaying a bitmap resource follows:

SysPostMessage(5,LCD_PRINT_RANGE_RSRC,CONNECT_LEFT_X,CONNECT_LEFT_Y,ConnectImageLeft);

where parameters are:

params, MESSAGE_TYPE,X coordinate, Y coordinate, Resource,ID

The first parameter (# params) is the number of total parameters including the first parameter. This is required as the SysPostMessage function can take a dynamic number of parameters depending on the MESSAGE_TYPE.

Below are details of the transition as well as information on what was changed between SDK 2.320 and 2.4xx.

SDK 2.320 Projects\SDK\lcdexample\dcc\dcclcd.asm	SDK 2.4xx Projects\SDK\lcdexample\usbmsc\usbmsc_lcd.c
DisplayInit	USBLCDDisplayInit()
DisplayUsbIdle	USBLCDIdle()
DisplayUsbReading	USBLCDReading()
DisplayUsbWriting	USBLCDWriting()
DisplayLowBattery	USBLCDLowBattery()

Table 32: USBMSC Display Functions

All of the above functions accomplish displaying images to the LCD using the SysPostMessage function. The images of interest have the following resource IDs and corresponding bitmap file located in the rsrc folder in the usbmsc project directory.

RESOURCE ID			Bitmap File
ConnectImageLeft	equ	1	connectleft.bmp
ConnectImageRight	equ	2	connectright.bmp
LowBatImage	equ	7	lowbat.bmp
NotesImage	equ	6	notes.bmp
RSRC_PERIOD	equ	10	period.bmp

Table 33: USBMSC Display Bitmap Resource Usage

ReadingImage	equ	4	reading.bmp
ReadyImage	equ	3	ready.bmp
RSRC_VERSION	equ	9	version.bmp
WritingImage	**equ**	**5**	writing.bmp

Table 33: USBMSC Display Bitmap Resource Usage

20.1.1. USBLCDDisplayInit

This function is the first display function that is called after the display hardware is initialized. It is responsible for setting up what is on the display in the very beginning. This includes the picture of a computer and a picture of an external media device. The versions major and minor are also displayed by this function. The main difference between this function and its corresponding function in SDK 2.320 is the usage of SysPostMessage as opposed to the CallLCD macro.

20.1.2. USBLCDIdle

This function is responsible for displaying the Ready Image as well as clearing any notes that may be on the screen. The notes are animated, moving to the left or to the right, depending on weather the STMP is sending or receiving data.

20.1.3. USBLCDReading

This function displays the Reading Image and also calls USBLCDShiftNotes() which animates the notes that are on the screen.

20.1.4. USBLCDWriting

This function displays the Writing Image on the screen and also animates the notes that are on screen. It is different from the Reading function in that it animates the notes on the screen in the opposite direction.

20.1.5. USBLCDLowBattery

This function displays an image to indicate to the user that the battery is low so that they have time to remove the device from the computer.

20.1.6. USB Throughput

The usbmsc firmware released with SDK 2.4xx has added a feature that will allow some performance measurements on the USB bus while sending and receiving.

Note: SigmaTel does not recommend using these numbers for public releases because of inaccuracies due to measurement techniques. These are included for development purposes only. Additional documentation will be provided at a later date describing the measurement techniques.

This requires that a build option be enabled in usbmsc.mk. To enable this option modify the following line in usbmsc.mk by removing the '#' at the beginning. This will uncomment the line making it active.

#BLD=$(BLD) -DUSB_THROUGHPUT

The data that will be shown on screen is as follows:

Coordinates		Data
X	Y	
0	16	Input Transfer Rate
0	24	Output Transfer Rate
0	40	Number of Common Commands
32	40	Number of Updater Commands
64	**40**	Number of Unknown Commands

Table 34: USBMSC Throughput Display Data

20.1.7. USB Battery Display

During the DisplayUsbIdle API, a battery display was added for SDK2.320. Therefore, the battery will not be updated during the write or read cycles. This functionality could easily be added to DisplayUsbReading and DisplayUSBWriting if desired.

The GetBatteryLevel() function is called to obtain the current battery reading. The level is then added to the 0% resource to select the correct bitmap.

GetBatteryLevel() will also set the low battery flag, but the actual low battery display will be handled with DisplayLowBattery API. The USB battery display will use the same defines (VBATT_MIN_MV, VBATT_MAX_MV, etc) that are assigned in the project.inc.

Note: The USBMSC does not use the global *project.inc* and has a local version within its folder (...\lcdexample\usbmsc), which requires the battery defines to be assigned in two locations.

20.1.8. Changing Battery Resources

If the number of battery resources are changed from the SDK total of 10, changes will be needed to ensure an error does not occur. In addition to updating the resource.inc and usbmsc.mk, the GetBatteryLevel() should return the battery level based on the number of resources.

20.2. GROUP6 VENDOR SPECIFIC SCSI COMMAND SUPPORT

USB Mass Storage Class is a wrapper around SCSI commands, which allow the host OS to read and write to the media as if it were a standard drive. SigmaTel has extended the Group6 Vendor Specific command series during the new firmware updater implementation. This group of commands has been extended to allow custom commands to be integrated into the standard SigmaTel SDK.

The host updater communicates with the device by sending a 16-byte command, which is processed by the USBMSC firmware. If the command does not match a Sigmatel specific function, the GetCustomerExtentionSCSIHandler() will be called to handle customer specific commands.

20.2.1. Data Structures

20.2.1.1. SCSI Command Data Structure

WORD pCdb →

Bit / Byte	7	6	5	4	3	2	1	0
0	WOpCode							
1	Updater Command ID							
2								
3	Second Word							
4								
5								
6	Third Word							
7								
8								
9	Fourth Word							
10								
11								
12	Fifth Word							
13								
14								
15								

- Struct Command{}; - Data structure for SCSI command. Bit fields are defined as unions to easily extract byte information.

```
struct Command{union FirstWord; union Second Word; union ThirdWord; union
FourthWord; union FifthWord};
```

- WORD *pCdb - Pointer to FirstWord of command stored in the buffer.
- WORD wOpCode - Variable containing the SCSI Opcode (SCSI_OPCODE_XXX). Customer OpCodes will need to be added.

```
//Sigmatel Specific Opcodes
#define SCSI_OPCODE_SIGMATEL_READ_COMMAND  0xc0
#define SCSI_OPCODE_SIGMATEL_WRITE_COMMAND 0xc1
```

- SCSI_COMMAND_ENTRY g_UpdaterCommands[] - Array containing updater command IDs and function addresses.

```
//Sigmatel Specific Updater Commands
#define GET_STATUS   //UpdaterGetStatus ID
SCSI_COMMAND_ENTRY g_UpdaterCommands[] =
{
  {GET_PROTOCOL_VERSION,      &UpdaterGetProtocolVersion},
  {GET_STATUS,                &UpdaterGetStatus},
...};
```

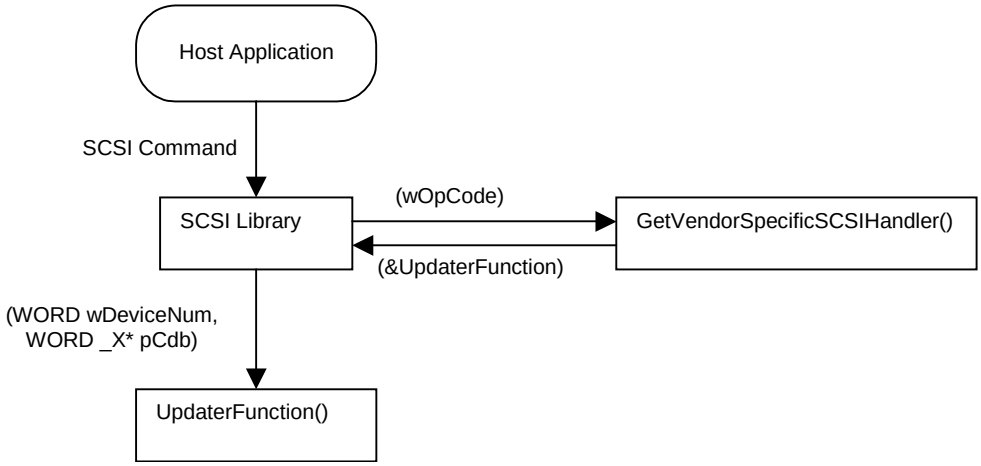
20.2.2. Adding SCSI Commands

The figure below shows a flow chart of how the SCSI commands will be processed. The host application will send the SCSI command. The SCSI library will fill the command

buffer and begin processing the OpCode. If the OpCode is vendor specific, GetVendorSpecificSCSIHandler() will determine which vendor function (UpdaterFunction) should be called and return its address to the SCSI Library. The SCSI task manager will then call the UpdaterFunction.

To handle new SCSI commands, OpCodes, Updater Command IDs, and Updater Functions will need to be added to the existing codebase. A skeleton command handler, GetCustomerExtentionSCSIHandler(), has already been added to process custom commands.

Figure 28. Flow Chart of Updater SCSI Command



20.2.3. Adding OpCodes

Vendor specific OpCodes will need be added for customized host commands. The new OpCodes must begin after the Sigmatel Opcodes to prevent conflicts with the SCSI standard or Sigmatel Commands. The host will need to ensure that the OpCode is placed in byte 0 of the Command buffer.

```
//Example Customer Opcodes
#define SCSI_OPCODE_CUSTOMER_COMMAND1  0xC2
#define SCSI_OPCODE_CUSTOMER_COMMAND2  0xC3
```

20.2.4. Adding Updater Commands

As shown in Figure 1 - SCSI Command Data Structure, byte 1 of the command buffer holds the Updater Command ID. The command IDs will be processed within GetCustomerExtentionSCSIHandler(). A new array could also be created to match the ID to the address of the corresponding updater function.

```
//Example Updater Command IDs
#define CUSTOM_TASK1 0x01
#define CuSTOM_TASK2 0x02

//Example Updater Command Array
SCSI_COMMAND_ENTRY g_UpdaterCustomCommands[] =
{
  {CUSTOM_TASK1,       &UpdaterTask1},
   {CUSTOM_TASK2,       &UpdaterTask2}
};
```

COMPANY CONFIDENTIAL 9/26/03

20.2.5. Adding Custom Bitfields to Command Structure

It may be desirable to create new bitfields within the Command structure to match the 16-btye host SCSI command. These bitfields can be used with the pCdb buffer pointer to extract individual bytes.

```
//Example Bitfield for FirstWord
struct Command{
  union {
       struct {
       WORD wSCSICustOpCode    : 8;
       WORD wCustomCommand     : 8;
       WORD wUnUsed            : 8;
  }Custom;
} FirstWord;
```

20.2.6. Command Handling with GetCustomerExtentionSCSIHandler()

The skeleton for GetCustomerExtentionSCSIHandler() has been added to the SDK, but the command handling will need to be included. There are two methods to process the commands. The first is to return the proper function address based on the Command ID. The second method is have the Updater Function act as the command handler similar to SigmatelSCSICommandHandler(). Examples of these methods are outlined in the following sections.

20.2.6.1. Method 1 - Return UpdaterFunction via SCSI Handler

The SCSI handler can be used to return the address of the updater function. For Method 1, the OpCode and Command ID will both be processed to select the correct Updater Function.

```
//Example SCSI Handler
SCSI_CH _reentrant GetCustomerExtentionSCSIHandler(WORD wOpCode)
{
    SCSI_CH pFunc=NULL;
    switch(wOpCode)  //Determine which OpCode has been sent
    {
       case SCSI_OPCODE_CUSTOMER_COMMAND1:
       // Determine which Command ID has been sent
       // select correct function address
        if(((struct Command _X*)pCdb)->FirstWord.Custom.wCustomCommand ==
CUSTOM_TASK1)
            pFunc = &UpdaterTask1;
       else if (((struct Command _X*)pCdb)->FirstWord.Custom.wCustomCommand ==
CUSTOM_TASK2)
            pFunc = &UpdaterTask2;
        break;

       case SCSI_OPCODE_CUSTOMER_COMMAND2:
        if(((struct Command _X*)pCdb)->FirstWord.Custom.wCustomCommand ==
CUSTOM_TASK1)
            pFunc = &UpdaterTask1;
       else if (((struct Command _X*)pCdb)->FirstWord.Custom.wCustomCommand ==
CUSTOM_TASK2)
            pFunc = &UpdaterTask2;
        break;

    default:
        pFunc  = NULL;  //Unknown OpCode
```

COMPANY CONFIDENTIAL 9/26/03


```
        break;
          }
      return pFunc;
      }
```

20.2.6.2. Method 2 - SCSI Handler as Updater Function

This is the method used for the SigmatelSCSICommandHandler(). The Vendor SCSI handler will return the address for Sigmatel Command Handler. The Sigmatel Command Handler will then be responsible for selecting and calling the proper updater function. The advantage of this method is to have more control over structures such as the SCSI_LUN (pLun).

The following example shows what edits are required in the Vendor SCSI handler and how to configure the Customer Command Handler.

```
//Edit Vender SCSI Handler to return.  Edited code in Red
SCSI_CH _reentrant GetVendorSpecificSCSIHandler(WORD wOpCode)
{
    SCSI_CH pFunc=NULL;
    switch(wOpCode)
    {
    case SCSI_OPCODE_SIGMATEL_READ_COMMAND:
    case SCSI_OPCODE_SIGMATEL_WRITE_COMMAND:
        g_iUpdaterCommands++;
        pFunc = &SigmalSCSICommandHandler;
        break;
    case SCSI_OPCODE_CUSTOMER_COMMAND1:
    case SCSI_OPCODE_CUSTOMER_COMMAND2:
        g_iUpdaterCommands++;
        pFunc = &GetCustomerExtentionSCSIHandler;
        break;
    default:
        pFunc = NULL;
        g_iUnknownCommands++;
        break;
    }
    return pFunc;
}
//Custom Command Handler based on the SigmatelSCSICommandHandler()
RETCODE _reentrant GetCustomerExtentionSCSIHandler(WORD wDeviceNum, WORD _X
*pCdb)
{
    WORD RetValue = 0;
    WORD wCommand;
    SCSI_CH pFunc;
    USBMSC_DEVICE * pDev;
    SCSI_LUN * pLun;

    pDev = &(UsbMscDevice[wDeviceNum]);
    pLun = &(pDev->Lun[pDev->CBW.wCBWLUN]);

    //set up default sense data.
    ScsiLunSetupDefaultSenseData(pLun);
    // Select Updater Function by sending Command ID Array and pointer
    // to Command ID to SearchForSCSICommand() in the SCSI Library.
    //  Returns Updater Function Address
   if((pFunc =  SearchForSCSICommand(g_UpdaterCustomCommands,((struct Command
```

```
    _X*)pCdb)->FirstWord.Custom.wCustomCommand)))
    {
        RetValue = (pFunc)(wDeviceNum,pCdb);//Call Updater Function
        if(RetValue!=SUCCESS)   //If it failed, Set-up status flags
        {
            pLun->wCompletionStatus = USBMSC_CSW_COMMAND_FAILED;
            pLun->SenseData.wSenseKey = SCSI_SENSE_KEY_VENDOR_SPECIFIC;
            pLun->SenseData.wAscqAsc = lcl_Status;
        }
    }
    else
    {
        //Shouldn't see any commands we don't support.
        RetValue = SCSI_ASC_INVALID_COMMAND_OPERATION_CODE;
    }
    return RetValue;
}
```

20.2.7. Adding Updater Functions

The Updater Functions will do the actual work. Depending on which method of command handling is used, the Updater Function may be called via the SCSI library or the Customer Command Handler. The updater function definition must conform to the SDK standard.

```
//Skeleton Updater Function
RETCODE _reentrant UpdaterTask1(wDeviceNum, *pCdb)
{
   wDeviceNum;  //Device number
   pCdb;  //pointer to command buffer
 //Add code here
return SUCCESS;
}
```

APPENDIX A. BUILDING BLOCKS

A-1. Build Execution

player.bat :

- calls mk2410 using environment variables to control build
- build can be controlled by makefile alone

player.mk:

- calls other system .mk files in …\etc\
- standard.mk controls suffix rules
- includes dependencies for source program file only
- rebuild ALL if change *.h/*.inc file
- project specific files and resources; some system files

A-2. Player Output Files

player.out : intermediate file, not located

player.abs : located binary file; contains code, symbols, overlaid code in virtual memory

player.map : USE IT…defines memory locations/sizes of all sections/files

player.sb : converts s-record created by extractcluster; Tasking s-rec file NOT compatible with bootloader

resource.bin : resources stored in binary file

A-3. Creating Resources

.src files created from .bmp .txt or .abs

.bmp and .txt display files converted to data resources (BuildRes)

sections/clusters from .abs converted to code resources (ExtractCluster)

*.loc files are input files defining sections required for a code resource

resource.bin created by rsclinker using resource.inc as input

resource.inc names resources & defines order and content of resource.bin

A-4. Mixed Assembly / C

Symbol Representation

Assembler : FVar

C: Var

A-5. Locator Labels : F_lc_

used to load resources

COMPANY CONFIDENTIAL 9/26/03

APPENDIX B. ADDING ANOTHER MENU MODULE

Adding a custom menu to your player can be a daunting task to the beginner but with the help of this document that misconception will be eradicated and you will be able to give your player the personal feel needed.

The best place to start is with menus that already exist. Such menus are the Main Menu, and the Settings Menu. Menus of this type are a list of text items that serve as gateways into other portions of the player. Each item in the list has a corresponding string resource associated with it. Once you have decided what text you would like to use for your menu item, create your resource and update the player.mak file and resources.inc file. The menu item for this example will be represented on screen by the text “Test Menu”. This particular menu item is going to appear in the main menu so a small change must be made to the MainMenu.h file that already exists. Changes made are marked in blue.

```
#ifndef _MAIN_MENU_H
#define _MAIN_MENU_H
// menus in mainmenu
#define MAINMENU_FIRST  0
#define MENU_MUSIC      0
#define MENU_VOICE      1
#define MENU_FMTUNER    2
#define MENU_SETTINGS   3
#define MENU_DELETE     4
#define MENU_ABOUT      5
#define MENU_MAIN_EXIT  6
#define MENU_TEST       7
#define MAINMENU_LAST   7
#define MAINMENU_COUNT  MAINMENU_LAST+1
#define MAINMENU_PAGE1_COUNT  4
#endif
```

In this case the menu item has been added to the end of the list and as such will be the last item displayed on screen. The order of menu item listings is arbitrary and is equivalent to using selection indexes in VB or html or VC++. When scrolling through items they will be scrolled in the order above, starting with item 0. It is important to note that the nomenclature used here is also arbitrary. You may call your menu item anything you would like unless the name is used by a variable or constant. Once you have added your item in the mainmenu.h file it is time to give your item some meaning. This requires adding to the mainmenu.c file. Every menu item in the main menu is represented by a C structure called MenuItem. Each menu, such as the main menu, stores these structures in an array called aItems[]. The first thing that we need to do is define what text will be displayed on screen for out menu item. This is done by storing a pointer to the resource created above in the structure for our menu item. Below is the code required to do this. All additions are in blue.

```
// load resource numbers
aItems[MENU_MUSIC].m_iResource = RSRC_STRING_MUSIC_MENU;
aItems[MENU_VOICE].m_iResource = RSRC_STRING_VOICE_MENU;
aItems[MENU_FMTUNER].m_iResource = RSRC_STRING_FMTUNER_MENU;
aItems[MENU_SETTINGS].m_iResource =
RSRC_STRING_SETTINGS_MENU;
aItems[MENU_DELETE].m_iResource = RSRC_STRING_DELETE_MENU;
aItems[MENU_ABOUT].m_iResource = RSRC_STRING_ABOUT_MENU;
aItems[MENU_MAIN_EXIT].m_iResource = RSRC_STRING_EXIT_MENU;
aItems[MENU_TEST].m_iResource = RSRC_STRING_TEST_MENU;
RSRC_STRING_TEST_MENU is the name used for our text resource
```

COMPANY CONFIDENTIAL 9/26/03


```
in the resource.inc file.  The next thing that needs to be
accomplished is setting up x and y coordinates for text
placement.  The MenuItem structure already has space allo-
cated for this so it is a matter of storing two numbers.
Below is a excerpt from the example code.
for (a = 0; a < MAINMENU_COUNT; a++)
    {   // m_ixPos is 0 for all menu labels -- start at same
X loc
         aItems[a].m_ixPos = 0;
    }
    // load Y position  m_iyPos -- each line is 8 pixels
lower
    aItems[MENU_MUSIC].m_iyPos = 10;
    aItems[MENU_VOICE].m_iyPos = 18;
    aItems[MENU_FMTUNER].m_iyPos = 26;
    aItems[MENU_SETTINGS].m_iyPos = 34;
    aItems[MENU_DELETE].m_iyPos = 10;
    aItems[MENU_ABOUT].m_iyPos = 18;
    aItems[MENU_MAIN_EXIT].m_iyPos = 26;
   aItems[MENU_TEST].m_iyPos = 34;
```

The for loop above takes care of setting up the x coordinates for all of the menu items. They can be set at anything that your LCD will allow, and they do not have to all have the same x coordinate. The y coordinates are handled next. Only so many items can fit on one screen so a constant was added to MainMenu.h to determine how many items will be displayed at a time. When the last item on one screen is reached if more items exist then the rest of the items will be loaded is further scrolling is done. Be careful not to draw your menu on top of system icons that may be on screen, this could lead to undesired visual effects. This leaves the last change to mainmenu.c. Somewhere in your code should be a switch case on the iSelectedItem variable. You must add in a case for your menu item be selected. Below is an excerpt from the example code.

```
switch(iSelectedItem)
                    {
                    case MENU_MAIN_EXIT:
                    case MENU_MUSIC:
                        g_iCurrentMenu  = MENU_MUSIC;
                        iResource = RSRC_MUSIC_MENU_CODE_BANK;
                        pMenu = MusicMenu;
                        break;
                    case MENU_TEST:
                        g_iCurrentMenu  = MENU_TEST;
                        iResource = RSRC_ABOUT_MENU_CODE_BANK;
                        pMenu = TestMenu;
                        break;
```

This part can be complicated due to the CODE_BANK statement above. A code bank is made for different .c files. To minimize the work involved in creating your menu, add your menu code into a .c file that already exists. With the example code we have chosen to put example code in the AboutMenu.c file as it is small and has room for expansion. Any .c file can be used however; care must be taken to not make your .c file too big. This could lead to errors later on. Lastly notice the pMenu assignment. This is a pointer to a function of our creation. Place this function in AboutMenu.c as intended by the CODE_BANK choice. The function name will be the name TestMenu. It is now time to discuss the addition of this function.

The first thing that must be done is the creation of a function prototype. The function will be prototyped in menus.h.

```
//Export all Menu prototypes for calls allowed from other
code banks
void _reentrant UserTask(int a, int b, int *pPtr);
_reentrant INT ShutdownMenu( INT iIgnored1, INT iIgnored2,
INT *pPtr);
int _reentrant MusicMenu(INT a, INT b, INT *c);
int _reentrant VoiceMenu(int a, int b, int *c);
int _reentrant SettingsMenu(int a, int b, int *pPtr);
int _reentrant EqMenu(int a, int b, int *pPtr);
int _reentrant TestMenu(int a, int b, int *pPtr);
int _reentrant PlayModeMenu(int a, int b, int *pPtr);
```

The new function is named TestMenu. The name of the function must match that of the pointer stored in pMenu. Now that the function has been prototyped it is time to create it. Since you have already decided that your function is going to be added to the about menu code bank you must put your function definition in AboutMenu.c. To make this easier use the code in `AboutMenu(int a, int b, int *pPtr)` as your code. A copy and paste will work. At this point you should have a function definition that looks like this:

```
int _reentrant TestMenu(int a, int b, int *pPtr)
{
    union DisplayHints DisplayHint;
    BOOL bRefreshDisplay;
    INT  iEvent;
    BOOL bDone = FALSE;
    INT  iNextMenu = MENU_MAIN;
    BOOL bSendToPlayerStateMachine = FALSE;
*************All of the other pasted code as
well***************
}
```

Now everything is in place and your menu item will essentially do something, however, it will do exactly what the about menu does so we should change it up...

```
while(!bDone)
{
  if(bRefreshDisplay)
  {
      SysCallFunction(RSRC_DISPLAY_CODE_BANK,RefreshDis-
play,DisplayHint.I,1,0);
      if (DisplayHint.bits.ClearDisplay)
      {  // if cleared display, redisplay info
         //*****Add Your Code Here!!!*********
      }
         //after displaying, clear all display hints
         DisplayHint.I=0;
        bRefreshDisplay = FALSE;
  }
}
```

All you need to do at this point is insert your code into the spot:

//*****Add Your Code Here!!!*********.

Your code can consist of anything you want including but not limited to the displaying of resources.

COMPANY CONFIDENTIAL 9/26/03

APPENDIX C. ADDING A NEW MENU CODEBANK

1. Add a string resource to the build

   Make a .txt file in rsrc folder (i.e. string_test_menu.txt). The contents of this file is a text string which will be displayed in the main menu.

   Add this string resource to the make file and resource.inc so it will be built automatically when the player is built.

   in resource.inc: `RSRC_STRING_TEST_MENU equ ### ;$FILENAME string_test_menu.src`

   in make file (RESOURCELIST): `$(OUTPUT_RSRC_PATH)\string_test_menu.src\`

   in make file(# Text Resources): `$(OUTPUT_RSRC_PATH)\string_test_menu.src : $(PROJRSRC_PATH)\strings\string_test_menu.txt`

2. Update mainmenu.h with your menu name. (i.e. `#define MENU_TEST some number`)
3. Also update *MAINMENU_LAST* in mainmenu.h
4. Update mainmenu.c

   aItems[MENU_TEST].m_iResource = RSRC_STRING_TEST_MENU;

   aItems[MENU_TEST].m_iyPos = 16; //y position is your choice as is the x

5. Add a case for your menu in: switch(iSelectedItem). **Cover in step 9 below.**

   Add a new CODE_BANK to the resource.inc file.

   Our .c file is going to be named testmenu.c. So the following is added: `RSRC_TEST_MENU_CODE_BANK equ ### ;$FILENAME testmenu.src`

6. Update the make file to accommodate this CODE_BANK

   Make addition to the CODEBANKLIST: $(OUTPUT_RSRC_PATH)\testmenu.src\

   Make addition to the OBJLIST: `$(OBJ_PATH)\testmenu.obj\`

   Make addition to # Project-specific files: This will be a dependency list for your .c file. Currently testmenu.c will only have one dependency, namely testmenu.h.: `$(OBJ_PATH)\testmenu.obj :$(PROJMENUS_PATH)\testmenu.c $(OBJ_PATH)\testmenu.obj :$(PROJMENUS_PATH)\testmenu.h`

   Make an addition to Module Code Extraction section: `"$(OUTPUT_RSRC_PATH)\testmenu.src" : $(OUTPUT_PATH)\$(PROJ).abs menus.loc`

7. Update make\menus.loc

   Add: `testmenu.src={MenuManager_p,.ptexttestmenu};`

8. Update make\menus.dsc

   Add: `section .ptexttestmenu MP3_COPY;`

9. Return to mainmenu.c where you added a case for the switch(iSelectedItem). And make it look like the following:

```
case MENU_TEST:
  g_iCurrentMenu = MENU_TEST;
  iResource      = RSRC_TEST_MENU_CODE_BANK
  pMenu    = TestMenu; //VoiceMenu;
  break;
```

COMPANY CONFIDENTIAL 9/26/03

Note: TestMenu is the name of the function that will be jumped to in testmenu.c. We must now setup that file.

10. Open menus.h

    Add: `int _reentrant TestMenu(int a, int b, int *pPtr);`

    Create testmenu.c and testmenu.h and open testmenu.c

    Add: `int _reentrant TestMenu(int a, int b, int *pPtr){}` function definition to the .c file.

APPENDIX D. ADDING A NEW MODULE

D-1. Module Table (modulemsg.asm):

Add module to module number list in the order to be executed

Increment module count

Add module entry to module table in same order as number list and initialize:

Event Signal: 0 if module is not active during system init, else EVENT_INIT

Event Wait: EVENT_INIT

Resource # of module code (same as added to resource.inc)

Message Queue Descriptor base address

Process Queue: name of routine in module code that processes messages

Initialization routine: name of routine in module code that initializes module called when EVENT_INIT is set in both Event Signal and Event Wait.

Timer High : 0

Timer Low : 0

Background : Empty Function - not used in current code base

Message Queue (modulemsg.asm):

Increment MSG_QUEUE_COUNT equate defining the number of modules

Create 2 equates defining the size in words of the message buffer and the modulo value (size -1)

Define a circular message buffer using the defined size

Define message queue descriptor

Router Table (modulemsg.asm):

Add entry for new module -- use message ID from (msgequ.inc); message queue descriptor; and module number

Add Module Resource (resource.inc):

Add module resource number equate. The filename must be the same as the resourcefilename.src in the makefile.

D-2. Add to the MakeFile (player.mak):

filename.obj to the macro list (7)

resourcefilename.src to the macro list (8)

dependency to build .obj (16)

dependency to convert .inc to .h if required (17)

extractcluster rule to build .src **uses cluster name from .dsc(19)

D-3. Add to the descriptor file (stmp3400.dsc):

Section name of module to block PModuleOverlayBlock and cluster P_Module_Overlay_clstr (9) -- if resource number is in a separate section (menus) it must be overlaid as the first word of the cluster

Build options for debugging overlays if using CrossView Debugger -- #ifdef to (9) and #ifndef to (10)

Cluster to hold copy of module in block XExtBlock or block YExtBlock (12) **the name of this cluster is used in the extractcluster makefile rule

COMPANY CONFIDENTIAL 9/26/03

APPENDIX E. BUILDING CUSTOM FONT RESOURCES

1. Download ActivePerl from http://www.activeperl.com/products. After the installation of ActivePerl, the path "C:\Perl\bin" will be added to system path.
2. Put "C:\your SDK directory\bin" directory in system path by following way: right click My Computer icon->Advanced->Environment Variables->Path->Edit. After adding the directory in system path, reboot PC to make it effective.
3. Create a directory for building your new font, such as "C:\CodeFont".
4. Copy a codepage file, such as iso8859.txt or cp936.txt, to you new directory.
5. Go to DOS command. Change path to the new directory, and type the command: "build_dbcs2unicode.pl YourCodePageFile" (e.g. "build_dbcs2unicode.pl iso8859"). This step will create three files below:
   - iso8859_unicodeused.txt,
   - iso8859_pdm.src,
   - iso8859_sdms.src
6. Unzip your font bitmap files to your new directory (e.g. sigmatel_8.zip).
7. Use the FontPrefix/bitmap file name(stmp1) in sigmatel_8.zip to rename file iso8859_pdm.src and iso8859_sdms.src to stmp1_pdm.src and stmp1_sdms.src.
8. If you want to change font size (stmp1 is 6x8 now) or create new font bitmap, just replace .bmp files with the right size fonts/bitmaps or add new bitmaps to the directory.
9. In the new directory, if there is no default file FontPrefix_0001.bmp (e.g. stmp1_0001.bmp), create the file as a default which will map any non-exist characters to this character.
10. Go to command windows again and change path to the new directory. Type following command: "build_font.pl FontPrefix CodePage_UnicodeUsed.txt" (e.g. "build_font.pl stmp1 iso8859_UnicodeUsed.txt").
11. Last step will create the following files:
    - stmp1_PGM.src
    - stmp1_SGMs.src
    - stmp1_scripts.src
    - stmp1_script_00.src
12. Copy following .src files to C:\ your SDK directory \projects\sdk\lcdexample\player\rsrc.
    - stmp1_pdm.src
    - stmp1_sdms.src,
    - stmp1_PGM.src
    - stmp1_SGMs.src
    - stmp1_scripts.src
    - stmp1_script_00.src

13. Find the section in the file C:\stmp3xxx_sdk2312\projects\sdk\lcdexample\player\resource.inc.

```
RSRC_PGM_8          equ   241    ;$FILENAME sigmatel_8_PGM.src
RSRC_SGMS_8         equ   242    ;$FILENAME sigmatel_8_SGMs.src
RSRC_SCRIPT_00_8    equ   243    ;$FILENAME sigmatel_8_script_00.src
RSRC_SCRIPTS_8      equ   244    ;$FILENAME sigmatel_8_scripts.src
RSRC_PDM            equ   245    ;$FILENAME sigmatel_8_PDM.src
RSRC_SDMS           equ   246    ;$FILENAME sigmatel_8_SDMs.src
```

Replace the above .src files with the files:

- stmp1_PGM.src
- stmp1_SGMs.src
- stmp1_script_00.src
- stmp1_scripts.src
- stmp1_pdm.src
- stmp1_sdms.src

14. If there is only one font is built into the firmware, there is not need to change font_table.src file. Go to last three steps directly.
15. If there are more than one font are built, repeat the step 6-12 using another FontPrefix name (e.g. stmp2. Suggest to create another directory, C:\CodeFont1 for second font build). Create:
    - Stmp2_PGM.src
    - Stmp2_SGMs.src
    - Stmp2_scripts.src
    - Stmp2_script_00.src
16. Add the following lines in resource.inc, and assign a resource number for each of the lines. Also renumber other line resource numbers following this section and make them continue:

```
RSRC_PGM_10          equ   245    ;$FILENAME stmp2_PGM.src
RSRC_SGMS_10         equ   246    ;$FILENAME stmp2_SGMs.src
RSRC_SCRIPT_00_10    equ   247    ;$FILENAME stmp2_script_00.src
RSRC_SCRIPTS_10      equ   248    ;$FILENAME stmp2_scripts.src
```

17. Edit C:\ your SDK directory \projects\sdk\lcdexample\player\rsrc\font_table.src. the file should look like following:

```
-----------------------------------
$RESOURCE_TYPE
DATA
$LABEL
font_tableData
$REC_SIZE
```

COMPANY CONFIDENTIAL 9/26/03


```
8
$DATA
00000f
000002
000001
#RSRC_PGM_8
#RSRC_SGMS_8
#RSRC_SCRIPT_00_8
#RSRC_SCRIPTS_8
-------------------------------------
```

18. Add four lines in the file just like below:

```
-------------------------------------
$RESOURCE_TYPE                1
DATA                          2
$LABEL                        3
font_tableData                4
$REC_SIZE                     5
12                            6
$DATA                         7
00001b                        8
000002                        9
000002                        10
#RSRC_PGM_8                   11
#RSRC_SGMS_8                  12
#RSRC_SCRIPT_00_8             13
#RSRC_SCRIPTS_8               14
#RSRC_PGM_10                  15
#RSRC_SGMS_10                 16
#RSRC_SCRIPT_00_10            17
#RSRC_SCRIPTS_10              18
-------------------------------------
```

19. The change should be made from line 6, 8, 9, and 10. The explanation is as following:

    Line 6: total number of lines after line 6 to the end line (here is 12).

    Line 8: is total number of bytes in hex from line 10 to the end of line (each line has three bytes. Here is 9x3=27=0x00001b).

Line 9: data type: 0x000001=string resource; 0x000002=data resource; 0x000003=bitmap resource; 0x000004 button resource.

Line 10: number of fonts; here is two since we have two fonts.

20. The font selection is decided by function call “SysPostMessage(3,LCD_SET_FONT,wFontIndex)” to switch font. The wFontIndex is font index in font_table.src. “0” is the first font or default font, and then “1” is the second font.
21. Define a GLOBAL variable g_iLanguageNum and refer to the following lines at the place to switch fonts:

```
switch(g_iLanguageNum)
{

    case LANGUAGE_CHINESE:
        SysPostMessage(3,LCD_SET_FONT,1);
        g_iSDMS = RSRC_SDMS_CP936;
        g_iPDM = RSRC_PDM_CP936;
        break;
    case LANGUAGE_KOREAN:
        SysPostMessage(3,LCD_SET_FONT,2);
        g_iSDMS = RSRC_SDMS_CP949;
        g_iPDM = RSRC_PDM_CP949;
        break;
    default://English
        SysPostMessage(3,LCD_SET_FONT,0);
        g_iSDMS = RSRC_SDMS_iso8859;
        g_iPDM = RSRC_PDM_iso8859;
        break;
 }
if (s_btHandle_PDM)// Close resource
{
    SysCloseResource(s_btHandle_PDM);
    s_btHandle_PDM = 0;
}
if (s_bthandle_SDM)// Close resource
{
    SysCloseResource(s_bthandle_SDM);
    s_bthandle_SDM = 0;
```

```
}
```

Where g_iPDM, g_iSDMS, s_btHandle_PDM, and s_bthandle_SDM are global variables defined in SDK.

The last three steps:

1. Edit player.mak if necessary. The makefile must be updated to include the new font path (i.e wherever the new .src files are placed).
2. Rebuild player
3. Update your firmware with the new resource.bin and stmpsys.sb.